Architecture and Performance Analysis of a Multi-Generation SDRAM Controller for Mixed Criticality Systems

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von Leonardo Luiz Ecco

aus Chapecó - Santa Catarina - Brasilien

eingereicht am 15.05.2018

mündliche Prüfung am: 12.07.2018

1. Referent: Prof. Dr.-Ing. Rolf Ernst
2. Referent: Prof. Dr.-Ing. Norbert Wehn

Druckjahr: 2018

Abstract

Since the beginning of the millennium, the semiconductor industry has promoted a paradigm shift in the design of microprocessor chips. The approach consisted on slowly replacing deep power-hungry pipelines by multi- and many-core platforms containing simpler processing cores. At the same time, the real-time community started focusing on how to implement mixed criticality support in such platforms. More specifically, on how a multi- or many-core system can be designed so that applications or functions of different criticality levels can safely co-execute.

A criticality (or a criticality level) can refer to all forms of dependability, but is mostly used in the context of functional safety, i.e. the absence of catastrophic consequences for the user and/or environment. As safety-critical functions are generally subject to timing requirements, most mixed criticality systems are also real-time systems. With regard to timing, which is one of the focuses of this dissertation, a common example of a mixed criticality system includes a *best-effort* level, for applications that demand good average performance, and a *critical* level, for applications that demand good worst-case timing bounds.

Implementing support to different classes of criticality in a multi- or many-core platform poses an interesting challenge because such platforms rely extensively on the use of shared resources. The advantage of sharing resources is economic: simply stated, sharing allows the final cost of the system to be reduced. However, sharing also allows different cores to interfere with the performance of each other. As a consequence, designing effective resource sharing mechanisms is crucial for the commercial success of a platform. Typical resources shared in multi- and many-core platforms include the cache, the interconnect fabric and the main memory. This dissertation focuses on the latter, more accurately on the design of the controller of the main memory.

Due to their high-density and low-cost, DDR SDRAMs are the prevailing choice for implementing the main memory of a computer system. Nevertheless, the aforementioned benefits come at the cost of a complex two-stage access protocol, which ultimately means that the time required to serve a memory request depends on the history of previous requests. Otherly stated, DDR SDRAMs are a *stateful* resource. Leveraging the *state* in order to provide good average performance for *best-effort* requestors without compromising guarantees for *critical* requestors is the main goal of this dissertation.

With that regard, this dissertation firstly identifies two challenges of growing relevance for the design of memory controllers for the mixed criticality domain. The first challenge is the data bus turnaround time. In SDRAMs, a single data bus is used for read and write operations. Hence, alternating the execution of read and write commands is highly

undesirable, as it takes time to reverse the direction of the data bus, i.e. the data bus turnaround time. In COTS systems, which are purely performance oriented, the challenge is tackled by buffering write commands and executing them in a batch once the number of write commands reaches a predetermined number. In mixed criticality systems, however, the same strategy is not acceptable, as it leads to hard-to-predict behaviors.

The second challenge is the rank-to-rank switching time and only affects multi-rank modules. A rank, in SDRAM jargon, refers to a set of chips operating under the same clock, chip-select and command bus. A multi-rank module refers to a printed-circuit board containing two or more ranks, which in turn share the same multi-drop data bus. Hence, alternating the control of the multi-drop data bus between different ranks also demands a predetermined amount of time, i.e. the rank-switching time, during which the data bus must remain idle. Having an idle data bus basically means the resource is not being utilized to its full potential, which is not a desirable feature. At the same time, any strategy to minimize rank switches should not lead to unbounded latencies for *critical* requestors.

After pinpointing the two aforementioned challenges, this dissertation proposes a SDRAM controller to tackle them. The proposed controller bundles *read* and *write* operations in their corresponding ranks, thus minimizing the number of data bus turnarounds and rank switching events. As a consequence, the average performance of the controller is improved. However, the bundling is carefully designed so that real-time guarantees for *critical* requestors can be extracted.

Moreover, as it will become clear, both the operation of the controller and the corresponding analysis of the temporal properties are described in terms of a generation-independent notation. This is a desirable feature because different SDRAM generations have different architectural features and possibly, timing constraints.

Finally, an extensive comparison with the related work is performed. Furthermore, trends in worst-case latency over DDR SDRAMs from different speed bins and generations are presented and thoroughly discussed.

Contents

1. Introduction

For a long period of time, the semiconductor industry has relied mostly on frequency scaling (backed by technology scaling) and in deepening power-hungry processing pipelines in order to keep up with performance improvements expected by users, application developers and system integrators. However, in the beginning of the millennium, such strategy started showing signs of exhaustion, as two challenges gained relevance: the power wall [117] and the ILP wall [86].

The former refers to the trend of consuming exponentially increasing power with each factorial increase of operational frequency of an integrated circuit. The latter refers to the increasing difficulty in exploiting Instruction-Level Parallelism (ILP), i.e. difficulty in finding enough parallelism inside a single instruction stream. In order to tackle both challenges, the answer found by the semiconductor industry was to promote a paradigm shift from single- to multi- (and many-)core platforms, which contain two or more processing cores sharing communication and memory resources inside the same silicon chip.

As such platforms became widespread and gained market share, the real-time community started devoting more attention to them. More specifically, on how they could be employed in mixed criticality setups. As discussed in [29], mixed criticality is a quality attributed to systems in which application functions of different criticalities share computation, memory and/or communication resources.

The concept of criticality can include all forms of dependability, which is defined in [10] as the ability of a system to deliver service that can be justifiably trusted[1]. Forms of dependability include reliability, which refers to the continuity of correct service, integrity, which refers to the absence of improper system alterations, and safety, which refers to the absence of catastrophic consequences on the user(s) and the environment. As pointed out in [29], the term criticality is mostly employed in the context of the last of the aforementioned forms, i.e. safety. Hence, informally speaking[2], an application or function is said to be safety-critical if its failure leads to catastrophic consequences.

With regard to safety, the authors from [29] also point out that most safety-critical functions are subject to timing requirements, which means most mixed criticality systems are also real-time systems. Such observation is important because the time dimension of mixed criticality is one of the focuses of this dissertation. With regard to it, a dual-criticality system might include a level for (time-)*critical* functions, which demand good

[1] The authors of [10] also provide an alternate definition of dependability: the ability of a system to avoid service failures that are more frequent and more severe than acceptable.

[2] For formal definitions of criticality, the reader is invited to consult safety standards such as IEC 61508 [52] and ISO 26262 [57].

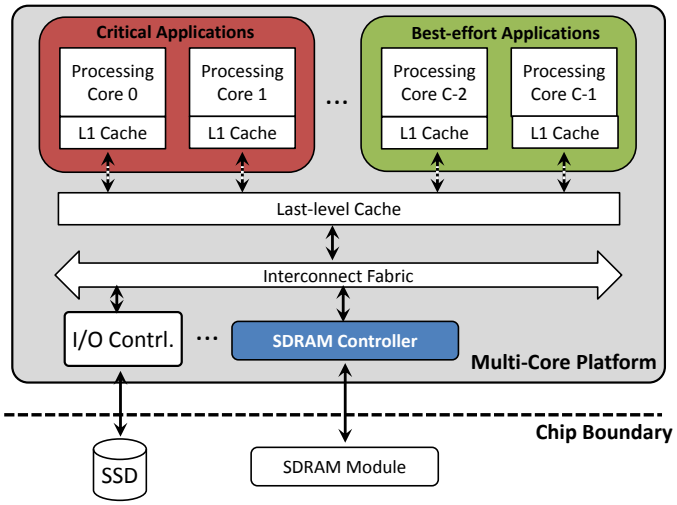


Figure 1.1.: Example of computing platform. The SDRAM controller, **main focus of this dissertation**, is depicted using the blue color.

worst-case timing bounds, e.g. an Anti-lock Braking System (ABS) or a traction control system, and *non-critical* (or *best-effort*) functions, which require no timing guarantees but benefit from good average performance, e.g. entertainment functionality.

For the sake of properly motivating the work performed within the scope of this dissertation, Fig. 1.1 depicts a didactic example of a multi-core platform in which the cores have been partitioned among applications of different criticalities. More specifically, a portion of the cores is dedicated to executing *critical* functions, while another portion is dedicated to executing *best-effort* functions.

The example displays two important main characteristics: the first one is the use of resource sharing, which includes among other resources one or more layers of cache memories, the interconnect fabric and the main memory controller, the main topic of this dissertation. The second one is the presence of a memory hierarchy, i.e. the platform relies on more than one type of memories. This chapter firstly provides a discussion about each of the characteristics and then highlights the contribution of this dissertation.

1.1. Resource Sharing and Mixed Criticality

Sharing one or more resources between multiple processing cores in a multi- or many-core platform is an effective strategy to reduce the cost of the final product. In the didactic example from Fig. 1.1, the last level of cache, the interconnect fabric and the SDRAM controller are shared among all processing cores of the platform.

However, although cost-effective, resource sharing allows different cores to interfere with each other both from the spatial and temporal perspectives. Hence, in a mixed criticality environment, resource sharing poses a threat to the safety of a system unless proper mechanisms to control the interference are deployed. With regard to interference

control, there are two possibilities: the first one is to simply design the entire system according to the highest criticality involved. This is, however, far too costly and, hence, unfeasible.

The second option, as dictated by safety standards such as IEC 61508 [52], requires *sufficient independence* between functions of different criticalities. This means that the system must be designed in a way that limits both the spatial and temporal interference that *non-critical* functions can exert on *critical* functions. As a consequence, *non-critical* functionality, e.g. infotainment software provided by a third party, does not need to go through a costly certification process, as even if it contain errors, such errors will not lead to a failure of *critical* components.

The remaining of this section firstly clarifies the criticality levels (and corresponding requirements) considered in this dissertation and then discusses which considerations must be made when designing mechanisms for safe resource sharing in mixed criticality setups.

1.1.1. Criticality Levels and Requirements

As discussed in the beginning of this chapter, this dissertation focuses mostly on the time dimension of mixed criticality. Consequently, the criticality levels discussed in this section refer to timing requirements.

This dissertation considers applications that belong to two different levels of criticality:

- (Time-)*Critical*, assigned to applications for which a worst-case bound on performance must be guaranteed, i.e. the behavior exhibited during run-time is predictable. The performance guarantee for *critical* applications is computed through a timing analysis, an analytical proof that takes into account the properties of the system, such as the scheduling implemented to share resources.

  It is important to observe that applications assigned to the (time-)*critical* level demand *predictable* service, i.e. worst-case bounds on performance. This is less restrictive than the concept of *composability* [41], which means full timing isolation. As in practice predictability is employed to create *composability* [6], the latter is not considered in this dissertation.
- *Best-effort*, assigned to applications do not demand guaranteed worst-case bounds on performance, such as entertainment functionality. Nevertheless, the user expects them to have good average performance. For instance, when an user clicks a button on a web browser, he/she expects the browser to react quickly.

1.1.2. Designing Resource Sharing Mechanisms

In order to design proper resource sharing mechanisms, a designer must take three main aspects into account, which are enumerated and discussed below:

1. *Knowledge about the resource itself.* Knowledge about the resource is necessary because it determines which type of considerations have to be made when implementing a sharing approach. As a rule of thumb, for resources that are *stateless*,

only timing considerations have to be made, while for resources that are *stateful*, the state (possibly in addition to timing) must be considered. For instance, sharing an interconnect bus depends simply on timing considerations, while sharing a cache depends mostly on spatial considerations. As it will become clear, SDRAMs fall into the *stateful* category.

2. *Goal of sharing.* Apart from decreasing the cost of the final product, implementing a mechanism for sharing a resource should be guided by the needs of the system. In a Commercial Off-the-Shelf (COTS) system, for instance, the ultimate goal when implementing a mechanism to share a resource is to improve the overall performance of the system. However, for the dual-time-criticality scenario considered in this dissertation, the goal is to provide good worst-case timing bounds for *critical* components, while at the same time providing good average performance for *best-effort* components. Notice that such goal is specially challenging because optimizing for worst-case and optimizing for average case are generally conflicting goals.
3. *Implementation strategy.* This refers to whether the mechanism is implemented in software, hardware or through a combination of both. In case of an external memory module, sharing is implemented in hardware within the memory controller. (Potentially, specific software might be deployed to control the access to the memory controller).

There is also a fourth aspect, which is mostly important for embedded systems that rely on batteries: power consumption. Ideally, an implementation strategy should consume as little power as possible, provided that the timing expectations of applications are fulfilled.

1.2. Memory Hierarchy and SDRAMs

As detailed in [58], a memory hierarchy is an approach to achieve the performance of fast memory devices at the cost per bit (and energy consumption) of cheap and low-power memory devices. More specifically, a carefully designed hierarchy exploits the advantages and hides the drawbacks from each memory technology. A simplified diagram of a memory hierarchy is depicted in Fig. 1.2. In the figure, notice that the larger the area of a layer, the larger the storage capacity such layer provides. For instance, a Solid State Drive (SSD) has a low cost per bit in comparison with the technology employed to implement caches and, hence, is more suited for mass storage roles (notice that such advantages come at cost of increased latency).

Still in the figure, notice that the main memory, whose **controller design is one of the main focuses of this dissertation**, is highlighted. The main memory is responsible for holding the code and data of the workload currently being executed by the processing cores of a platform. In summary, the code and data from applications are copied from permanent storage devices into the main memory either during boot-up (or at arbitrary instants), after which the processing cores can execute them. (To be highlighted is also the fact that, during execution, main memory locations that are likely

to be frequently accessed are stored into one or more layers of cache memories).

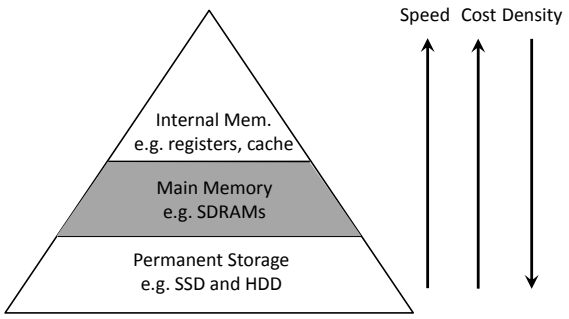


Figure 1.2.: Simplified diagram of memory hierarchy (based on what is depicted in [58]).

Currently, the prevailing choice for implementing the main memory of computing platforms are Double Data Rate (DDR) Synchronous Dynamic Random Access Memories (SDRAMs) [60, 61, 62] (as non-DDR SDRAMs are outdated, the DDR prefix will be from now on omitted). Such memories are synchronous because they operate with a clock, dynamic because information is stored in capacitors (which lose their charge if not refreshed), and are said to have double data rate because data is transferred in both rising and falling clock edges. The main characteristics that make SDRAMs a suitable choice for the role of main memory are their high-density and cost-effectiveness.

The rest of this section firstly discusses SDRAMs and SDRAM controllers, as they constitute the **main focus of this dissertation**. Then, it provides a discussion about system-level considerations, which are necessary because not only SDRAMs (and their controllers) are shared between multiple cores in a multi- or many-core platform, but also a large portion of the memory hierarchy.

1.2.1. SDRAMs

In order to deploy SDRAMs in a system, two aspects must be addressed: the first is command scheduling and the second is physical communication.

From the command scheduling perspective, designers must consider the clearly defined interface of SDRAMs which is described in the corresponding standards established by the JEDEC Solid State Technology Association [60, 61, 62]. More specifically, each SDRAM device contains a set of banks, as depicted in Fig. 1.3. Each bank contains a matrix of data (built using capacitors) and an intermediate level of caching called the *row buffer*.

Cells in a data matrix of a bank are only accessible through the corresponding *row buffer*. In order to read or write data from/into cells of a bank, the entire row that contains the desired cells must firstly be loaded into the corresponding *row buffer*, which is accomplished using the *activate* command. Then, values stored in cells can be retrieved using the *read* command or overwritten using the *write* command. Finally, if cells from a different row need to be retrieved or overwritten, the contents of the *row buffer* must

firstly be written back into its data matrix, which is accomplished using the *precharge* command.

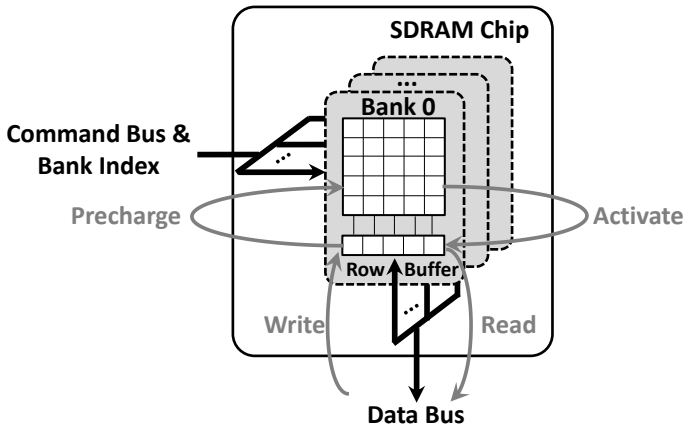


Figure 1.3.: Simplified structure of SDRAM chip and commands used to transfer data. In the figure, the data and command buses are depicted in black, while the movement of data triggered by the execution of commands is depicted in gray. Notice that the figure does not show the *refresh* command, which is not related to data transfers and whose purpose is to prevent the capacitors that hold data from being discharged.

It is important to notice that although each bank enjoys a certain degree of independence from other banks in the system, all banks share the command bus and the data bus. Consequently, in a single clock cycle, it is not possible to execute different commands for two different banks in the system. (Notice that each SDRAM chip has a set of input ports which are used to indicate the target bank of the command available in the command bus. Such input ports are referred to as *bank index* in the figure.).

With regard to the aforementioned 2-stage access protocol, two challenges are highlighted: 1) Commands cannot simply be executed back to-back. More specifically, there are timing constraints that dictate a minimum distance between any two consecutive commands. In practical terms, such constraints mean that SDRAMs have a *stateful* nature, in which the response time of a request (which demands commands) depends on the history of previous requests. 2) Each new generation of SDRAM devices introduces new architectural features and/or timing constraints. Consequently, approaches to abstract specific features and/or constraints are interesting, as they allow a scheduling algorithm to be evaluated over SDRAM devices from different generations.

From the physical communication perspective, it is important to observe that, as depicted in Fig. 1.1, SDRAMs are generally not manufactured within the chip that contains the processing cores of a system. Consequently, in order to use SDRAMs, system designers assemble a System-in-a-Package (SiP), which contains the chip with the processing cores plus SDRAM chips (or SDRAM modules). The reason for it is that manufacturing them together is not cost-effective [76], as manufacturing processes

for logic circuits and for (dynamic) memories have conflicting needs: for the former, high-speed is the main goal, while for the latter, high-density and low leakage are more desirable.

Because of their complex access protocol and the fact that the interface between processing cores and SDRAMs crosses the chip boundary, the logic required to control the flow of data between the processing cores and the main memory is implemented in a digital circuit called an *SDRAM controller*. The SDRAM controller is responsible not only for handling physical-level communication details, but also for serializing incoming requests which arrive through the interconnect fabric (potentially through multiple input ports).

1.2.2. SDRAM Controller Design

A SDRAM controller is responsible for serving SDRAM requests. A read request is basically a 2-tuple containing an address, from which data must be read, and a size, the amount of data that should be read. A write request is a 3-tuple: in addition to an address and size, it also contains the chunk of data which shall be written into the memory. In order to clarify how requests are processed by a SDRAM controller, a simplified block diagram of a SDRAM controller is depicted in Fig. 1.4.

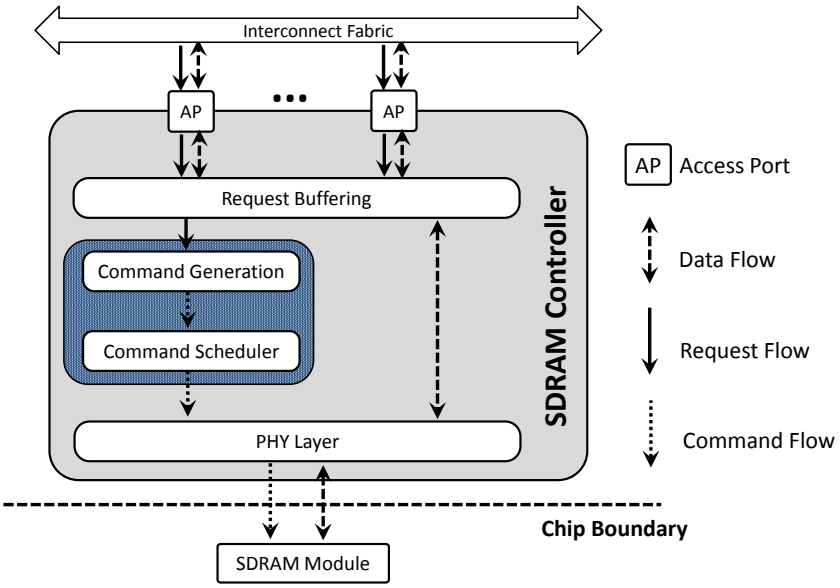


Figure 1.4.: Simplified diagram of a multi-port SDRAM controller. The portions that are the focus of this dissertation are highlighted in blue.

Incoming requests arrive through the interconnect fabric and enter the SDRAM controller through one or more input ports. Requests are then stored into request buffers. Once in the buffers, they become eligible for execution. For that purpose, they firstly go through a command generation layer, which translates them into the proper set of SDRAM command sequences. Then, the command scheduling layer arbitrates between

multiple pending commands (that possibly target different SDRAM banks) and forwards the winner into the PHY layer. The PHY-layer executes the commands it receives from the scheduling layer, i.e. it implements circuit-level communication details, e.g. sampling data in the correct rising clock edges after a *read* command is executed. Again, such layer is necessary because the controller and the SDRAM module are not implemented within the same chip. Finally, after the data transfer (or transfers) necessary to fulfill a request are performed, the controller notifies the processing core that issued the request that such request has been successfully served. Notice that, for read requests, the notification contains the data that was just read from the memory module, which is not the case for write requests.

The design of the command generation and scheduling is guided by the requirements of the workload. With regard to it, the author from [41] distinguishes between three categories of controllers, which are also pertinent for this dissertation:

1. Purely real-time controllers, whose main goal is to provide good worst-case timing bounds. For this category of controllers, only optimizations that improve the predictability of the controller are considered. Anything that cannot be guaranteed through a formal timing analysis is deemed as useless.
2. Best-effort controllers, whose main goal is to maximize the average case performance. For this category of controllers, only optimizations that reduce the average latency of a request and increase throughput are considered useful, even if such optimizations lead to unpredictable behavior.
3. Mixed criticality controllers, which must be able to fulfill the requirements of both *best-effort* and *critical* applications. For this category, optimizations that improve average performance are only useful if they do not significantly worsen the guarantees that are provided for real-time applications.

Finally, it is important to observe that the logic that implements command generation and scheduling is orthogonal to the PHY layer, i.e. they are developed independently. More specifically, the PHY layer does not define the category of a SDRAM controller (among the three aforementioned ones). Consequently, this dissertation focuses on command generation and scheduling (and not on the PHY layer).

1.2.3. System-Level Considerations

As is the case for the example depicted in Fig. 1.1, multi- and many-core platforms generally share a large portion of the memory hierarchy, which includes one or more levels of cache, the interconnect fabric (e.g. a bus, a ring or a Network-on-Chip (NoC)) and permanent storage devices. (Technically speaking, the interconnect fabric is not part of the memory hierarchy. However, it is through such fabric that different levels of the memory hierarchy are connected.)

Hence, in order for the system to support mixed criticality, having only the SDRAM controller aware of it does not suffice, as timing interference can occur at any shared resource. Consequently, mechanisms to support the sharing of the interconnect fabric and the cache must also be deployed. Such mechanisms are orthogonal to the SDRAM

controller design and, hence, are out of the scope of this dissertation. With that regard, however, the interested reader can consult [47, 54, 16, 74, 115] for real-time considerations about the interconnect fabric and [19, 79, 120, 78] for considerations about caches. Moreover, it is to be highlighted that permanent storage devices have received little attention from the mixed criticality perspective because they are only accessed at boot time (or when an application needs to be loaded into the main memory).

1.3. Research Objectives and Contribution

In the two previous sections, the motivation for this dissertation has been outlined:

- The ILP wall and the power wall have forced a paradigm shift from single- to multi- and many-core platforms in the semiconductor industry.
- There is a trend in the real-time community to integrate software components of different criticalities onto the same multi- or many-core platform.
- SDRAM are the prevailing choice for main memory of a computer platform.
- As SDRAMs have a complex access protocol and are off-chip memories, the functionality required to operate them is implemented within a SDRAM controller.
- Each new generation of SDRAMs introduces new architectural features and/or timing constraints.
- For embedded-systems that rely on batteries, low power consumption is an important feature of a system.

This leads to the definition of the **main objective** of this dissertation: *to design a multi-generation SDRAM controller for multi- and many-core platforms with support for mixed criticality.*

As a **secondary objective**, the following is defined: *to evaluate worst-case bounds, average performance and power consumption trends that the proposed controller and the controllers from the related work display over SDRAM devices from different speed bins and generations.*

To fulfill such goal, the following steps are taken:

- First, from the perspective of SDRAM command scheduling, two challenges of growing relevance are pinpointed: the data bus turnaround time and the rank-switching overhead. The former affects commands executed in the same SDRAM chip and refers to the minimum timing interval between the execution of a *read* and of a *write* command (or vice-versa). Such time interval is necessary to change the On-Chip Termination (OCT) of a SDRAM device from input to output (or vice-versa). The latter affects multi-rank modules and refers to the minimum timing interval between consecutive transfers initiated by different SDRAM ranks. Such interval is required to enforce signal integrity in the multidrop data bus employed in mult-rank modules.

  As it will become clear, the data bus turnaround time and the rank-switching overhead decrease the ability of a controller to keep the data bus of a SDRAM

device (or module) occupied. Consequently, such effects damage worst-case timing bounds, relevant for *critical* software components, and average performance, relevant for *best-effort* software components.

- Second, as already mentioned, each new SDRAM generation introduces new architectural features and timing constraints. Consequently, in order to reason about command scheduling and timing analysis in a generation-independent fashion, a generic and flexible notation to refer to SDRAM timing constraints is presented.
- Third, a SDRAM controller that minimizes data bus turnarounds and rank-switching events is proposed. For that purpose, the controller employs read/write bundling. More specifically, it serves batches of *read* or *write* commands in a rank (before switching to another rank or performing a turnaround). Both the command scheduling and the corresponding timing analysis are described in terms of the aforementioned flexible notation for timing constraints.
- Lastly, a thorough evaluation is performed. The evaluation is divided into three main parts: the first part is an assessment of the influence of core-to-SDRAM-bank assignments. The second part is a comparison of the proposed controller with the related work from the worst-case, average-case and power consumption perspectives. And the third part is an evaluation of worst-case performance across SDRAM devices from different speed bins and generations.

This dissertation is structured as follows:

- Chapter 2 discusses SDRAM devices, SDRAM modules and timing constraints. Moreover, it provides a thorough discussion about data bus turnaround and rank-switching events, along with presenting a generic and flexible multi-generation notation to refer to timing constraints. Chapter 2 corresponds to the first and second steps taken to fulfill the research objectives.
- Chapter 4 describes a SDRAM controller architecture for mixed criticality systems. Along with Chapter 5, Chapter 4 corresponds to the third step taken to fulfill the research objectives.
- Chapter 5 presents a timing analysis of the controller, which is important for *critical* software components. Along with Chapter 4, Chapter 5 corresponds to the third step taken to fulfill the research objectives.
- Chapter 6 presents a thorough evaluation of the proposed controller controller and corresponds to the fourth step taken to fulfill the research objectives.
- Finally, Chapter 7 presents the concluding remarks.

2. Background on SDRAMs

This chapter firstly discusses SDRAM devices and SDRAM modules. Then, it describes a generic and flexible notation to refer to SDRAM timing constraints. Such notation is employed throughout this dissertation and can be used by other work in the field of SDRAM controllers.

2.1. SDRAM Devices

This section is structured as follows: firstly, it discusses naming conventions employed to identify SDRAM devices. Then, it describes the internal structure of SDRAM devices (along with their commands and corresponding timing constraints). Finally, it provides a thorough discussion about data bus turnarounds.

2.1.1. Naming Conventions for SDRAM Devices

As discussed in the introduction, this dissertation focuses on DDR SDRAMs. In general, manufacturers of such memories identify them with a string that uses the following pattern: *DDRx-(speed bin)(grade)*. The x stands for the generation, e.g. DDR2 or DDR3. The speed bin is represented using the theoretical peak data rate measured in MT/s (mega transfers per second), which corresponds to 2 times the frequency of the data bus measured in MHz (because of the double data rate). For instance, a DDR3-800E device is able to perform up to 800 MT/s and its data bus frequency is equal to 400 MHz. The letter appended to the end of the string distinguishes between devices from the same speed bin that have different timing constraints (the closer to 'A' the grade is, the smaller the constraints). For instance, a DDR3-800D device has smaller timing constraints than a DDR3-800E device, even though both operate with a data bus frequency of 400 MHz.

2.1.2. Internal Structure, Commands and Timing Constraints

Fig. 2.1 depicts the logical structure of a generic SDRAM device with a 1-bit data bus. An SDRAM device is divided into banks. Although banks in an SDRAM device enjoy a certain degree of independence from each other, they all share the same command bus and the same data bus. The exact number of banks in an SDRAM device varies across different generations, with possible values being 4 or 8 for DDR2, 8 for DDR3, and 8 or 16 for DDR4. For DDR4, the banks are further divided into *bank groups* of 4 banks, which are discussed later.

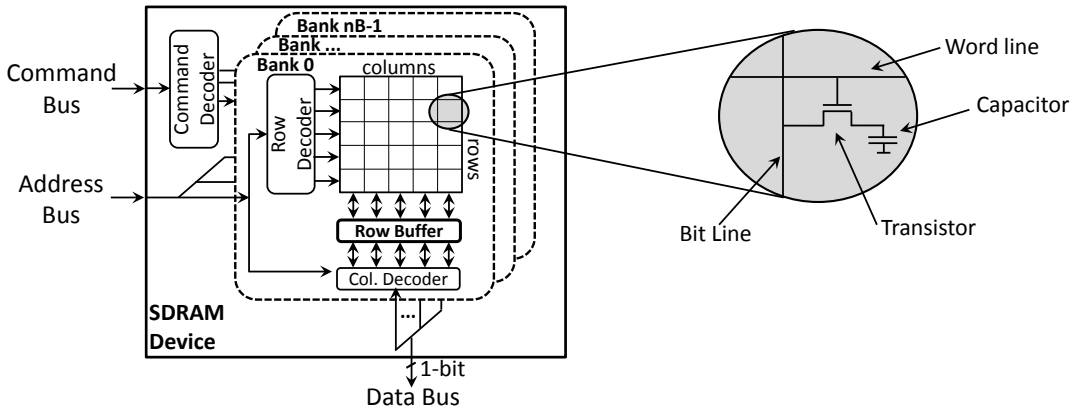


Figure 2.1.: Internal structure of generic SDRAM device with a 1-bit data bus. In devices with more than a single data bus pin, each bank would have more than a single capacitor array.

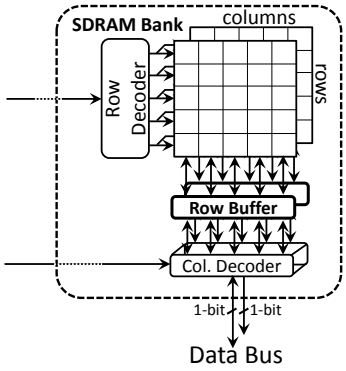


Figure 2.2.: An SDRAM bank with 2 capacitor arrays and a 2-bit data bus.

Each bank contains one or more capacitor arrays and a *row buffer*[1]. A cell in each array of capacitors contains exactly 1 capacitor, which stores a single bit of information. In Fig. 2.1, each bank has a single array and, as a consequence, the data bus width is 1. In SDRAM devices with a multi-bit data bus, e.g. 2, 4 or 8, each bank contains at least 2, 4 and 8 capacitor arrays, respectively. For the sake of clarity, Fig. 2.2 depicts an SDRAM bank that contains 2 arrays and a 2-bit data bus. (Notice that the figure is only depicting a single bank. Such bank would share the 2-bit data bus with other banks in the same device.)

All capacitor arrays of a bank operate in lock-step. As a consequence, from the logical perspective, one can think of the set of capacitor arrays in a bank as a data matrix, in which each of the cells has a data word whose width matches the width of the data bus.

[1]The *row buffer* is actually a set of differential sense amplifiers. This dissertation, however, focuses on the logical structure of SDRAMs, instead of their physical structure. For the latter, the interested reader can consult [58].

For instance, each cell in the data matrix of the bank depicted in Fig. 2.2 has two bits (and each of the two capacitor arrays in the bank contributes with one bit for the matrix cell). This data matrix abstraction is used throughout the rest of this dissertation.

As discussed in the introduction, the values stored in a cell of a data matrix are not directly visible to the SDRAM controller. All data exchanges are instead performed through the corresponding *row buffer*, which represents an intermediate level of caching between the controller and the SDRAM device. There are four commands used to move data into/from a *row buffer*: *activate* (A), *precharge* (P), *read* (R) and *write* (W). There is also a fifth command, which is not related to data transfers: the *refresh* (R). A detailed discussion about such commands is provided below:

- The *activate* command loads a matrix row into the corresponding *row buffer*, which is known as *opening a row*. More specifically, the *activate* command causes the word line to be asserted. Consequently, the circuit between the corresponding capacitors and bit lines is closed, which allows the values stored in the cells of the *activated* row to be propagated into the *row buffer*. Notice that before loading a row into a *row buffer*, the SDRAM controller must firstly *precharge* such *row buffer*.
- The *precharge* command writes the contents of a *row buffer* back into the corresponding matrix, which is known as *closing a row*.
- The *read* and *write* commands are used to retrieve or forward words from or into a *row buffer*. The acronym CAS (Column Address Strobe) is used to refer to both *read* and *write* commands and the letter (C) is used to refer to a CAS command. More specifically, if a pending command is certainly a *read* or a *write* (and not an *activate*, *precharge* or *refresh*), the text will simply refer to it as a CAS[2].

  CAS commands operate in bursts, which means that each of them transfers more than one word (from the SDRAM device into the controller in case of a *read* and in the opposite direction in case of a *write*). The exact amount of words transferred by a CAS command is determined by the the burst length (BL) parameter. A burst length of 8 words is supported by all DDR families investigated in this dissertation. A single CAS command occupies the data bus for $t_{BURST} = BL/2 = 4$ cycles and transfers $BL \cdot W_{BUS}$ bits, where W_{BUS} represents the width of the data bus. In this dissertation, systems with $BL = 8$ and $W_{BUS} = 64$ bits are considered, in which a single CAS command transfers 64 bytes (a common cache line size). Finally, it is important to observe that all banks in a SDRAM device share the same data bus. Hence, it is not possible to perform two different data transfers at the same time.
- The *refresh* command must be executed regularly[3] in order to prevent the capacitors that hold data from discharging [13]. According to the DDR2, DDR3 and

[2] In the JEDEC standards [60, 61, 62] and in [58], the acronym CAS is employed to refer to *read* commands. In the context of SDRAM controllers in time-critical systems, the acronym was firstly used to refer to *read* and *write* commands in [100], after which other articles, e.g. [77], started adopting the convention.

[3] Interesting cases in which explicit refreshing of rows can be omitted are discussed in [65].

DDR4 standards, the capacitors of a SDRAM row can hold their charge for at most 64 milliseconds[4], after which data loss can potentially happen[5]. Notice, however, that a single *refresh* command does not refresh all rows of a bank. For instance, in DDR3 [61], a total of 8192 *refresh* commands have to be executed within a 64 millisecond window and each of them refreshes a total of $N/8192$ rows, where N refers to the number of rows in a bank.

SDRAM timing constraints are now discussed. The documents that specify the DDR2/3/4 standards [60, 61, 62] describe in detail how each command changes the state of a SDRAM device and the time interval required for such changes to be performed. However, as described in [38, 41], from the perspective of the SDRAM controller, those details can be abstracted into timing constraints that dictate a minimum distance between consecutive SDRAM commands. Such constraints are measured in terms of data bus clock cycles and are enumerated in Table 2.1.

In the table, notice that several cells are marked as *not applicable* (n.a.). Those refer to command combinations that are either invalid or unconstrained. For instance, if cmd_a is a *write* then cmd_b cannot be an *activate* (in the same bank), because the corresponding *row buffer* would have to be firstly precharged. The other values present in the table cells refer to labels of the timing intervals required for command-triggered changes in a SDRAM device to complete. Except for t_{BURST}, which is defined as the duration of a data transfer, all other labels are employed in the DDR2/3/4 specifications[6].

For DDR4 devices, notice that some of the timing interval labels have a x suffix, which indicates that the value depends on whether cmd_a and cmd_b target banks that belong to the same *bank group* or not. If that is the case, the intervals are longer. In the DDR4 specification, the difference is identified by suffixing the corresponding labels with L (from long) or S (from short). For instance, consider two consecutive *activate* commands to different banks. If the banks are part of the same *bank group*, then the commands must be executed at least t_{RRD_L} cycles apart. However, if the banks are not part of the same *bank group*, they must be at least t_{RRD_S} cycles apart. Moreover, $t_{RRD\_L} > t_{RRD\_S}$.

Notice also that some constraints are the same regardless of whether cmd_a and cmd_b target the same bank or not. This is the case for any two consecutive CAS commands, which include not only the minimum distance between two *writes* or two *reads*, but also the data bus turnarounds mentioned in the introduction (cells in which $cmd_a = W$ and $cmd_b = R$ or vice-versa).

Table 2.1 is, however, not comprehensive. More specifically, there are two constraints

[4]For temperatures below 85°C.

[5]Several articles have measured the actual retention times of SDRAM devices [45, 122, 83, 84]. In summary, the results show that most rows are able to hold their data for longer periods than the ones specified in the standards. The authors in [98] even propose an approach to take advantage of such characteristic in order to avoid unnecessarily refreshing SDRAM rows.

[6]The DDR2/3/4 specifications employ the acronyms WL (write latency) and CWL (CAS write latency) interchangeably. The same observations apply to RL (read latency) and CL (CAS latency). Table 2.1 sticks to the WL and RL acronyms.

Table 2.1.: Minimum timing interval between cmd_a and cmd_b for three different generations of DDRx SDRAM. Extracted from [60], [61] and [62].

SDRAM Gen.	cmd_a	cmd_b=P Same bank	cmd_b=P Diff. bank	cmd_b=A Same Bank	cmd_b=A Diff. bank
DDR2	A	t_{RAS}	n.a.	t_{RC}	t_{RRD}
DDR2	P	n.a.	1	t_{RP}	n.a.
DDR2	R	$t_{BURST} - 2 + \max(t_{RTP}, 2)$	n.a.	n.a.	n.a.
DDR2	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.
DDR3	A	t_{RAS}	n.a.	t_{RC}	t_{RRD}
DDR3	P	n.a.	1	t_{RP}	n.a.
DDR3	R	$\max(t_{RTP}, 4)$	n.a.	n.a.	n.a.
DDR3	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.
DDR4	A	t_{RAS}	n.a.	t_{RC}	t_{RRD_x}
DDR4	P	n.a.	1	t_{RP}	n.a.
DDR4	R	t_{RTP}	n.a.	n.a.	n.a.
DDR4	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.
SDRAM Gen.	cmd_a	cmd_b=R Same bank	cmd_b=R Diff. bank	cmd_b=W Same Bank	cmd_b=W Diff. bank
DDR2	A	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR2	P	n.a.	n.a.	n.a.	n.a.
DDR2	R	t_{CCD}		4 if $t_{BURST} = 2$, else 6	
DDR2	W	$t_{RL} - 1 + t_{BURST} + t_{WTR}$		t_{CCD}	
DDR3	A	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR3	P	n.a.	n.a.	n.a.	n.a.
DDR3	R	t_{CCD}		$t_{RL} + t_{BURST} + 2 - t_{WL}$	
DDR3	W	$t_{WL} + t_{BURST} + t_{WTR}$		t_{CCD}	
DDR4	A	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR4	P	n.a.	n.a.	n.a.	n.a.
DDR4	R	t_{CCD_x}		$t_{RL} + t_{BURST} - t_{WL} + t_{PREAMBLE}$	
DDR4	W	$t_{WL} + t_{BURST} + t_{WTR\_x}$		t_{CCD_x}	

that are not represented in Table 2.1: t_{RFC} and the t_{FAW}. The t_{RFC} refers to the amount of cycles required for a *refresh* command to complete, e.g. $t_{RFC} = 36$ cycles for a DDR3-800E. The t_{FAW} constraint establishes a time window in which at most 4 *activate* commands can be executed (the acronym FAW stands for *Four Activation Window*). For the sake of clarity, a graphical depiction of t_{FAW} is provided in Fig. 2.3. Notice that $t_{FAW} > 4 \cdot t_{RRD}$.

Finally,the reader should be aware that the values of the constraints are larger for devices from faster speed bins. This is because the internal timing of different SDRAM devices varies little (mostly the data bus clock is increased). Hence, the timing interval that corresponds to the constraints measured in data bus clock cycles is larger for devices with high-speed data buses.

To illustrate the phenomenon, Fig. 2.4 (whose subfigures span over the boundary of a single page) depicts the time required to *precharge* and *activate* a *row buffer* in SDRAM

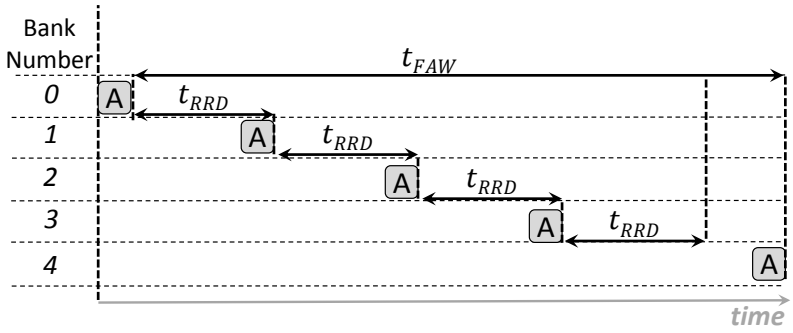
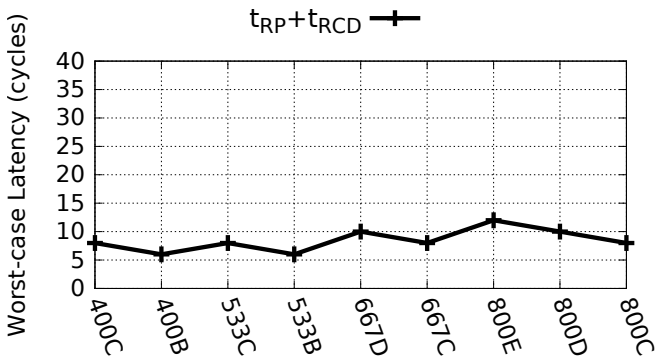


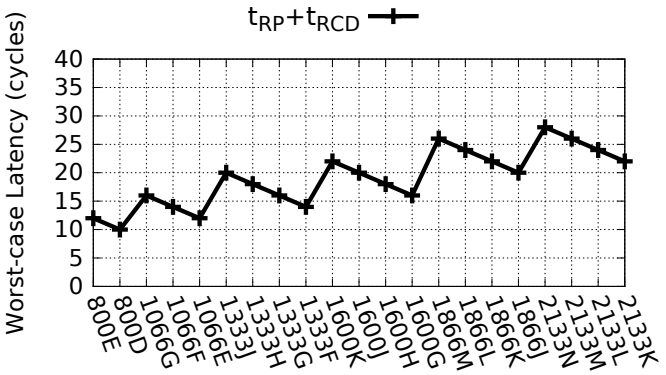
Figure 2.3.: The t_{FAW} constraint in a hypothetical device with a total of 5 banks (and no *bank groups*).

devices from different generations and speed bins. Using the notation employed in the DDR SDRAM standards [60, 61, 62], such time is equivalent to $t_{RP} + t_{RCD}$. Other examples of the aforementioned phenomenon are given in the next subsection.

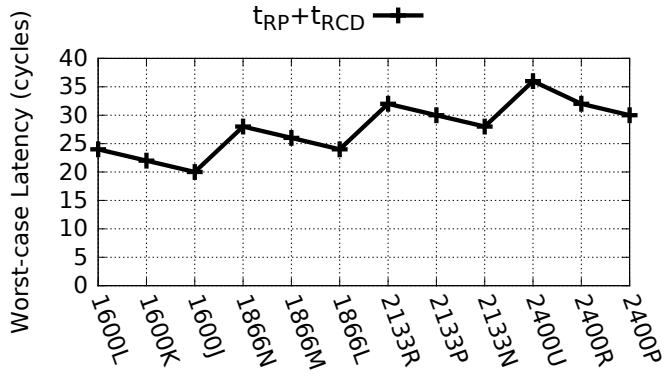


(a) DDR2.

Figure 2.4.: Number of data bus clock cycles required to *precharge* and *activate* a *row buffer*. (Continues in next page.)



(b) DDR3.



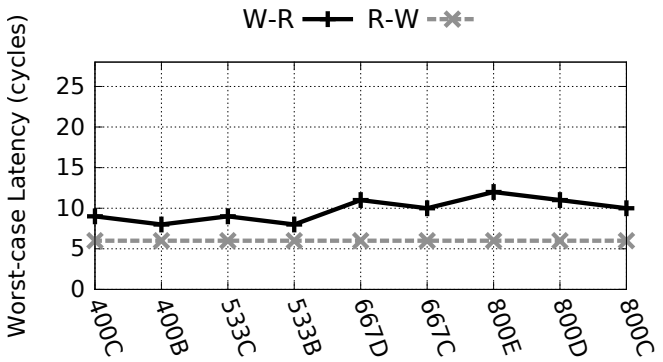
(c) DDR4.

Figure 2.4.: Number of data bus clock cycles required to *precharge* and *activate*.

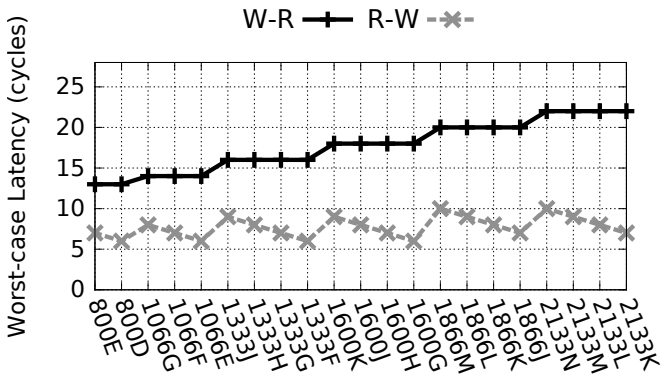
2.1.3. Data Bus Turnarounds

The data bus turnarounds constitute an important challenge in the design of effective SDRAM controllers for mixed criticality systems. In order to clarify the reason for it, Fig. 2.5 depicts the minimum distance (measured in data bus clock cycles) between the execution of a *read* command followed by a *write* (R-W) and of a *write* command followed by a *read* (W-R). Such distances represent the overhead for data bus turnarounds.

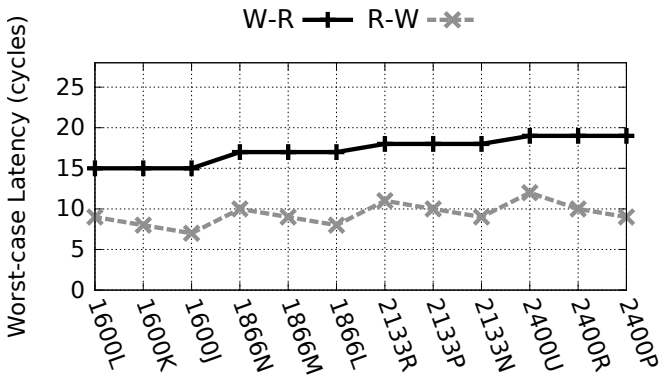
Notice that Fig. 2.5 is divided into four sub-figures: Fig. 2.5a considers that the pairs of commands are executed in DDR2 devices. Fig. 2.5b considers that the pairs of commands are executed in DDR3 devices. Fig. 2.5c considers that the pairs of commands are executed in banks belonging to the same *bank group* of DDR4 devices (short version of constraints). Fig. 2.5d considers that the pairs of commands are executed in banks belonging to different *bank group* of DDR4 devices (long version of constraints).



(a) DDR2.

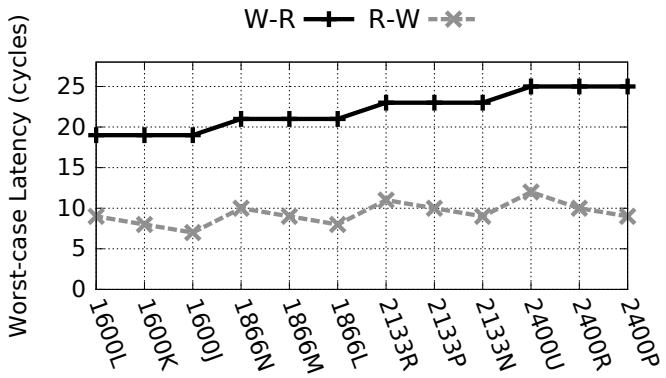


(b) DDR3.



(c) DDR4 (different bank group).

Figure 2.5.: Penalty for data bus turnarounds in DDR Devices. (Continues in next page.)



(d) DDR4 (same bank group).

Figure 2.5.: Penalty for data bus turnarounds in DDR Devices.

From the figures, the following observations are made:

- The overhead for turnarounds is smaller for a *read* that is followed by a *write* (R-W) than for a *write* that is followed by a *read* (W-R).
- Considering devices from the same generation and speed bin, the overhead depends on the grade of the device (as discussed in Section 2.1.1, the closer to 'A' the grade is, the smaller the timing constraints and, hence, the faster the device).
- Considering only the best grade of each speed bin of a generation (or only the worst grade), the overhead for turnarounds depends on the data bus clock frequency of a device. More specifically, the faster the data bus clock frequency, the larger the overhead for turnarounds. The reason for it, as discussed in the previous subsection, is that the internal timing of different SDRAM devices varies little (mostly the data bus clock is increased). Hence, the overhead measured in data bus clock cycles is larger for devices with high-speed data buses.
- For DDR4, the overhead is larger if the set of commands is executed in banks of the same *bank group*.
- As the overhead for turnarounds is mostly larger than the amount of cycles required to perform a single data transfer ($t_{BURST} = 4$), frequent turnarounds prevent a high utilisation of the data bus. For instance, in a DDR3-2133N device, the penalty for a R-W turnaround is 10 cycles and the penalty for a W-R turnaround is 22 cycles. Consequently, a controller that alternates the execution of *read* and *write* commands will only keep the data bus occupied with transfers for $\frac{2 \cdot t_{BURST}}{10+22} \cdot 100 = 25\%$ of the cycles (assuming no *refreshes*).

From the perspective of the design of SDRAM controllers for mixed criticality systems, the aforementioned observations should lead to the following conclusions: when performing command scheduling, a controller should minimize the number of data bus

turnarounds, thus increasing data bus utilisation (and improving the average latency of requests from *best-effort* applications). However, any mechanism to minimize turnarounds must not damage the worst-case bounds on the latency of requests from *critical* applications.

2.2. SDRAM Modules

This section discusses SDRAM modules and the rank-switching overhead.

Individual SDRAM devices have narrow data bus widths, e.g. 4, 8 or 16 bits. Hence, they are usually grouped under the same clock and chip-select signal into a so called SDRAM rank. From the perspective of the SDRAM controller, a SDRAM rank works exactly as a single SDRAM device, but contains a larger number of data bus pins and increased storage capacity. An SDRAM module is nothing more than a set of one or more ranks mounted on top of a Printed Circuit Board (PCB).

The physical structure of SDRAM modules has been standardized by JEDEC, e.g. in [63] and other documents. Types of SDRAM modules include Single In-line Memory Module (SIMM), Dual In-line Memory Module (DIMM) and Small Outline Dual In-Line Memory Module (SO-DIMM) (a compact version of DIMM, suited for environments in which space is a constraints, such as laptops). In SIMMs (which by now are outdated), the electrical contacts in both sides of the PCB are redundant, while in DIMMs and SO-DIMMs, they are not. This dissertation focuses on standard DIMMs with a total of 64 data bus pins, as they are commonly employed in many-core platforms, e.g. [102, 103].

A detailed discussion about the physical characteristics of DIMMs is available at [58]. Such discussion is important when designing the PHY layer of a SDRAM controller. However, as discussed in Section 1.2.2, the implementation of mixed criticality at the level of a SDRAM controller depends on command scheduling and not on the PHY layer. Consequently, this section focuses on the logical implications that the use of single- or multi-rank DIMMs have on the former. In order to aid the to-be-presented discussion, a generic dual-rank SDRAM module is depicted in Fig. 2.6.

The following observations about the figure are made:

- Each rank is composed of 8 8-bit SDRAM devices. As a consequence, each rank is seen by the controller as a single SDRAM device with a 64-bit data bus.
- Other configurations to build 64-bit ranks are also possible. For instance, using 4 16-bit SDRAM devices.
- The number of *rank banks*, i.e. the number of banks inside a rank, is equal to the number of *device banks* in a single SDRAM device. Hence, if the SDRAM devices used to form a rank have 4 banks, so will the corresponding rank. However, each *rank bank* (and corresponding *row buffer*) is x times larger than each *device bank*, where x is the number of SDRAM devices used to build a rank.

  For instance, considering the configuration from Fig. 2.6, each *rank bank* has 8 times the storage capacity of a *device bank*. Moreover, each *row buffer* of a *rank bank* is 8 times larger as a *row buffer* from a *device bank*.

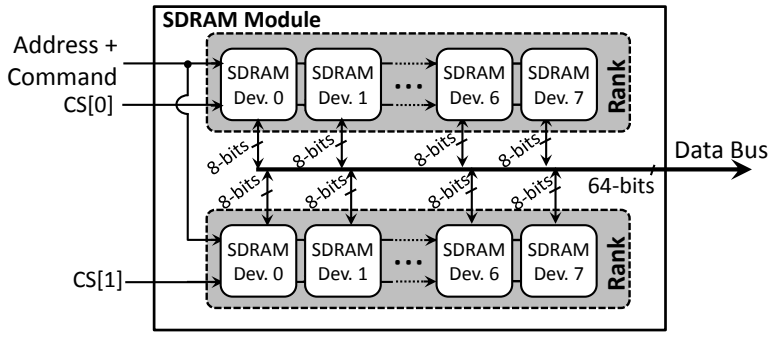


Figure 2.6.: Simplified diagram of a generic dual-rank SDRAM module.

- The main reason for employing multi-rank SDRAM modules is that each rank provides extra banks and, consequently, more storage space and a larger number of *row buffers*.
- A secondary reason for employing multi-rank SDRAM modules is that each rank is mostly independent from other ranks. More specifically, the timing constraints discussed in Section 2.1.2 only apply within a single rank (and, hence, are from now on referred to as *intra-rank* constraints). However, as the command, address and data buses are shared between all ranks, the scheduling of commands in multi-rank setups is also subject to *inter-rank* constraints.

The *inter-rank* constraints are now discussed. In order to execute a command in an specific rank, a SDRAM controller must assert the chip-select signal of such rank, deassert the chip-select signal of the remaining ranks, and then drive the command and address buses accordingly (for *write* commands, the data bus also must be driven). From the perspective of command scheduling, this means that at most one command can be executed per cycle, regardless of in which rank. (Notice that this is also the case if a single SDRAM device or single-rank module is considered.)

Furthermore, the minimum distance between consecutive CAS commands executed in different ranks must be larger than 1-cycle. The reason for it is that CAS commands trigger data transfers, which take place in a single data bus shared among all ranks. Notice that unlike the address and command buses, the data bus has multiple drivers (i.e. it is a multidrop bus). For a write operation, the SDRAM controller drives the data bus, while for a read, the corresponding rank does it (in order to send data back into the controller).

Hence, when scheduling CAS commands among different ranks, the controller must avoid collisions in the data bus, i.e. avoid that two or more ranks attempt to drive the data bus simultaneously. Moreover, the controller must enforce an idle timing interval between consecutive data transfers initiated by different senders so that the integrity of the electrical signals being transmitted is enforced. This timing interval is known as the *rank-to-rank switching time* or *rank-switching overhead* and, in this dissertation, is referred to as t_{RTRS}.

The t_{RTRS} value depends on the combination of SDRAM module and SDRAM controller and is characterized experimentally. For instance, AMD processors include a *PHY micro-controller unit* (PMU) which performs the characterization during boot [3] and then configures the on-chip SDRAM controller accordingly. Intel processors provide a similar mechanism [55]. In both AMD and Intel, the register used to store the *rank-to-rank switching time* has 4 bits. Hence, their on-chip SDRAM controllers expect up to 16 idle data bus cycles because of the rank-switching overhead.

From the perspective of command generation, it is useful to think of t_{RTRS} in terms of the impact it has on the minimum distances between consecutive CAS commands executed in different ranks. More specifically, as detailed in [25], t_{RTRS} leads to the so-called *inter-rank* timing constraints enumerated in Table 2.2. For ease of understanding, a graphical depiction of such inter-rank timing constraints is provided in Fig. 2.7.

Table 2.2.: Inter-rank timing constraints.

SDRAM	cmd_a	$cmd_b = R$ (executed in diff. rank)	
Gen.		Notation used in [25]	Computed as
DDR2/3/4	R	t_{RDRD_dr}	$t_{BURST} + t_{RTRS}$
	W	t_{WRRD_dr}	$t_{WL} - t_{RL} + t_{BURST} + t_{RTRS}$
SDRAM	cmd_a	$cmd_b = W$ (executed in diff. rank)	
Gen.		Notation used in [25]	Computed as
DDR2/3/4	R	t_{RDWR_dr}	$t_{RL} - t_{WL} + t_{BURST} + t_{RTRS}$
	W	t_{WRWR_dr}	t_{BURST}

This dissertation assumes a t_{RTRS} of 4.5 nano seconds, as it was reported in [119]. Notice, however, that SDRAM controllers measure time in terms of data bus clock cycles and, consequently, such value must be discretized. For instance, for a DDR3-800E module, the data bus is clocked at 400 MHz, i.e. a clock period of 2.5 ns, and hence $t_{RTRS} = \lceil \frac{4.5}{2.5} \rceil = 2$ cycles. For a DDR3-2133N, the data bus is clocked at 1066 MHz, i.e. a clock period of 1.07, and hence $t_{RTRS} = \lceil \frac{4.5}{1.07} \rceil = 5$ cycles.

As a single CAS command occupies the data bus for $t_{BURST} = BL/2 = 4$ cycles, a controller that blindly alternates between ranks can suffer from significantly low data bus utilisation. For instance, for a dual-rank DDR3-2133N, alternating *read* commands between different ranks means that the data bus will only be occupied during $\frac{t_{BURST}}{t_{BURST}+t_{RTRS}} \cdot 100 = 44\%$ of the cycles (assuming no *refreshes*). Although this is higher than the utilisation observed in a single-rank system that suffers consecutive turnarounds (see Section 2.1.3), it can still have a significant negative impact in performance.

Hence, as it was the case for data bus turnarounds, a SDRAM controller for mixed criticality systems should also minimize the number of rank-switching events (thus increasing data bus utilisation and improving the average latency of *best-effort* applications). However, any scheduling decisions that minimizes rank switches must not damage the worst-case bounds the latency of requests from *critical* applications.

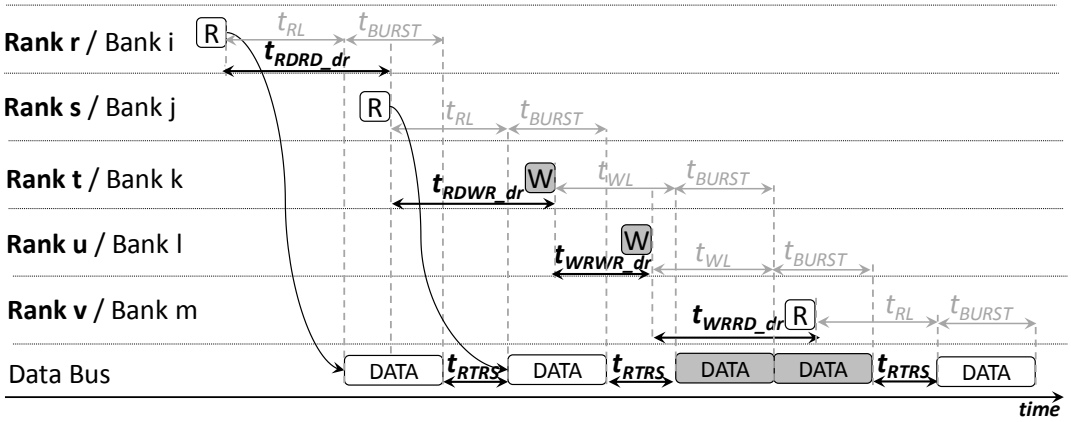


Figure 2.7.: Graphical depiction of inter-rank timing constraints in a hypothetical module with 5 ranks. Notice that two consecutive *write* transfers to different ranks do not demand a t_{RTRS} interval between them. This is because the same sender, i.e. the SDRAM controller, initiates both transfers.

2.3. Generic Notation for SDRAM Timing Constraints

In Sections 2.1 and 2.2, two types of timing constraints were discussed: *intra-rank* (or *intra-device*), which are a consequence of architectural features of SDRAM devices, and *inter-rank*, which only affect multi-rank SDRAM modules and are a consequence of two or more ranks sharing the data bus. These constraints were enumerated in Tables 2.1 and 2.2, respectively.

As discussed in [38], the tables represent (in practical terms) a *distance-function*. Such function receives as arguments the pair of commands under analysis and three Boolean values which determine respectively whether the pair of commands target the same rank, *bank group* and bank. Assuming that the DDR generation is implicitly given, the *distance-function* can be defined as follows:

$$d(cmd_a, cmd_b, sameRank, sameGroup, sameBank) \tag{2.1}$$

where:

- cmd_a and $cmd_b \in$ {A, P, R, W}
- $sameRank,\ sameGroup,\ sameBank \in$ {Yes, No}

Using such function, each cell from Tables 2.1 and 2.2 can be uniquely identified. For instance, *d(W,R,No,No,No)* refers to the minimum distance between the execution of a *write* and the execution of a *read* in a different rank. Notice that such representation is well suited to describe algorithms, e.g. the ones used to statically generate the command *patterns* in [38]. However, if it were to be employed in equations in a timing analysis, the same representation would lack clarity.

Hence, this dissertation proposes to represent any *d*-function invocation with the *d* prefix followed by a list of up to 5 arguments. The list of arguments employs the following

syntax:

- The first two arguments are mandatory and define the pair of commands under consideration, which are represented by the letters A, P, R and W.
- The pair of commands is separated from the list of Boolean arguments with a hyphen.
- Asserted Boolean arguments are identified as: R (for *sameRank*), G (for *sameGroup*) and B (for *sameBank*). Notice that the hyphen in the previous item eliminates the ambiguity caused by the letter R representing both a *read* command and the *sameRank* argument.
- Non-asserted Boolean arguments are identified as: $\overline{R}$, $\overline{G}$ and $\overline{B}$.
- *Don't care* values for Boolean arguments are represented by omitting them.

For instance, *dPA-RGB* represents the minimum distance between a *precharge* command followed by an *activate* command in the same rank, *bank group* and bank. Similarly, *dWR-*$\overline{R}$ represents the minimum distance between a *write* command followed by a *read* command in a different rank. For ease of comprehension, several examples of constraints using the proposed notation are depicted in Fig. 2.8.

Notice that, unlike *sameRank* and *sameBank*, the *sameGroup* argument is not employed to actually index a cell table. Instead, it works as a cell value modifier for the DDR4 generation. For instance, *dAA-RG*$\overline{B}$ and *dAA-R*$\overline{GB}$ refer to the minimum distance between two consecutive *activate* commands to different banks in the same rank. In Table 2.1, two commands fitting such description must be at least t_{RRD_x} cycles apart. If the *sameGroup* argument is true, i.e. *dAA-RG*$\overline{B}$, the long version of the constraint (t_{RRD_L}) is meant. If the *sameGroup* argument is false, i.e. *dAA-R*$\overline{GB}$, the short version of the constraint (t_{RRD_S}) is meant. In systems that do not have the *bank groups* feature, e.g. DDR2 and DDR3, the group argument is simply ignored. As it will become clear, this powerful abstraction allows a designer to provide a single design and performance analysis for a SDRAM controller, independently of SDRAM generation.

Last, three important remarks about the proposed notation are made:

- firstly, notice that it lacks an expression to describe the distance between the execution of a *read* (or *write*) command and the start of the corresponding data transfer. For that purpose, *dRD* (*read* to data) and *dWD* (*write* to data) are used. Examples of both are given in Fig. 2.8. In the DDR2/3/4 standards, the former refers to t_{RL} and the later to t_{WL}.
- Secondly, the expression *dCC* (potentially followed by a list of arguments) is used to refer to the minimum distance between any two consecutive CAS commands of the same type. For instance, *dCC-RG* refers to *dRR-RG* or *dWW-RG*.
- There is one constraint which cannot be represented using the proposed notation: t_{FAW}, which represents a time window in which at most 4 *activate* commands can be executed within a rank. Hence, the timing analysis in Chapter 5 simply refers to it as t_{FAW}.

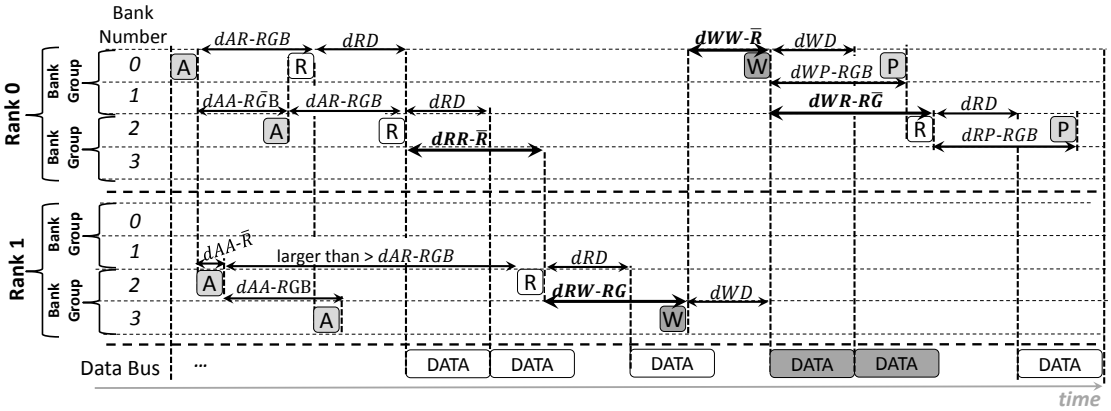


Figure 2.8.: Example of constraints using the proposed notation in a hypothetical DDR4 dual-rank module with 4 banks per rank divided into two *bank groups*.

2.4. Summary

This chapter discusses the foundation necessary to understand the operation of SDRAM devices and modules, which in turn is needed in order to understand how SDRAM controllers are designed.

The discussion is structured into 4 sections. The first section describes SDRAM devices, the commands used to operate them and the corresponding SDRAM timing constraints (*intra-device* or *intra-rank* constraints). Moreover, it also highlights one of the command scheduling challenges mentioned in the introduction: the overhead for data bus turnarounds, which refers to the minimum distance between pairs of CAS commands of different types (i.e. *read* followed by a *write* or vice-versa). As thoroughly discussed, such overhead can severely decrease data bus utilisation, having negative consequences in both worst-case and average performance. Hence, a SDRAM controller for mixed criticality systems should minimize turnaround events. However, any scheduling approach to achieve such goal must not damage the worst-case bounds on the latency of requests from *critical* applications.

The second section discusses how individual SDRAM devices are grouped together into SDRAM ranks, which in turn are mounted on a PCB in order to create a so-called SDRAM module. Furthermore, the section also highlights the second command scheduling challenge mentioned in the introduction (only pertinent for multi-rank modules): the rank-switching overhead, which refers to the minimum distance between consecutive data transfers initiated by different senders (with regard to this matter, a sender can be either the SDRAM controller or a SDRAM rank). In practical terms, the consequence of the rank-switching overhead is materialized in form of *inter-rank* timing constraints. Due to such constraints, a SDRAM controller that blindly alternates CAS commands over different ranks will suffer from low data bus utilisation. Hence, similarly to the case of data bus turnarounds, a SDRAM controller for mixed criticality systems should minimize rank switches, as long as any scheduling approach to do so does not damage the

worst-case bounds on the latency of requests from *critical* applications.

The third section discusses a flexible and multi-generation notation to refer to SDRAM timing constraints. The notation is useful because it allows designers to reason about SDRAM command scheduling and timing properties in a generation-independent fashion, which is specially important because every new SDRAM generation introduces new architectural features and/or new timing constraints.

The fourth and last section is this summary itself. Its purpose is simply to help the reader to consolidate his/her understanding of the presented content. Throughout this dissertation, every chapter is closed with a summary.

3. Related Work on SDRAM Controllers

This chapter discusses the related work on SDRAM controllers. Firstly, it presents a short overview of techniques employed in average-performance-oriented controllers. Then, it discusses controllers for real-time and mixed criticality environments. Finally, it describes the distinctive features of the controller proposed in this dissertation.

Before delving into the discussion, however, the reader should be aware that a large number of articles addresses the use of software techniques in the context of SDRAM controllers. Those include (but are not limited to) monitoring and throttling the number of requests that an application can issue [131, 130], performing SDRAM bank partitioning [64, 128, 85, 53] and reverse-engineering scheduling parameters of a controller [46]. Such approaches are not the focus of this dissertation and, hence, are not discussed here.

3.1. Average-Performance-Oriented Controllers

This category of SDRAM controllers focuses on increasing data bus utilisation and improving the average-latency of SDRAM requests without any regard for real-time considerations. For that purpose, optimizations that can lead to hard-to-predict behavior are employed. Those include (but are not limited to) the use of the First-Ready, First-Come First-Served (FR-FCFS) policy [105, 106, 67] and the buffering and opportunistic serving of write requests [129].

The FR-FCFS policy consists in exploiting *row buffer* locality by prioritizing (within the boundary of a single bank) the oldest request that targets the row currently stored in the corresponding *row buffer*. As a consequence, the number of *activate* and *precharge* operations that the controller executes is reduced. Hence, a potential performance improvement can be observed. However, such approach allows more recent requests to be served before older requests, which makes it difficult to compute a safe upper bound on memory latency without making assumptions about the behavior of all requestors that compete for a bank [67].

The buffering and opportunistic serving of write requests minimizes the number of data bus turnaround events, which are discussed in Section 2.1.3. The approach consists in buffering write requests until either no pending read request is available or the number of buffered write requests reaches a certain threshold. When one of the two aforementioned conditions is fulfilled, the buffered write requests are served in a batch. This avoids a granular interleaving of *read* and *write* commands, thus increasing data bus utilisation and improving average request latency. However, from the real-time perspective, it demands assuming a system backlogged with write requests in order to compute worst-case bounds on the latency of a request, which leads to poor timing bounds [129].

Apart from the two aforementioned approaches, a large number of articles address other optimizations for average-performance-oriented SDRAM controllers. For instance, in [56], the authors discuss a controller that uses reinforcement learning to adapt the scheduling policy during run-time. In [112, 111], the authors investigate the integration of SDRAM controller and the last-level of cache, showing that it can be beneficial both in reducing average latency of requests and in reducing power consumption. Moreover, [89, 90, 69, 68, 88] discuss controllers that improve overall throughput and increase fairness between requestors. Notice, however, that the aforementioned list is not comprehensive because average-performance-oriented controllers are not the focus of this dissertation.

3.2. Real-Time and Mixed-Criticality Controllers

A large number of articles proposing SDRAM controllers for real-time and mixed criticality environments have been proposed. They can be divided into two main groups: pattern-based and non-pattern-based. The former relies on precomputed command patterns that are scheduled during run-time. The latter generates commands dynamically and relies on no precomputed patterns.

Within the two groups, it is possible to further classify controllers using other criteria, which are listed and discussed below:

- **Row-policy**: also known in the literature as page-policy[1], the row-policy employed by a controller determines whether the *row buffers* are *precharged* between consecutive requests or not. There are two main types of row-policy: *close-row*, which means that the *row buffer* (or *row buffers*) used to serve a request are *precharged* before the controller processes the next request, and *open-row*, which means that *row buffers* are left in an open state after a request is served, i.e. they are only *precharged* if a *refresh* must be executed or if a pending request targets a row not currently present in the corresponding *row buffer*.

  There are also two variations of the two main types of row-policy, namely the conservative *open-row* policy [37] and the *mixed-row* policy [48]. Both are discussed in Section 3.2.1.
- **Request-to-bank mapping**: refers to how a request is mapped to the banks of a SDRAM device or module. There are two main types of mapping: *interleaved* and non-*interleaved*. In an *interleaved* mapping, each request accesses two or more banks of the SDRAM. In a non-*interleaved* mapping, each request accesses a single bank of the SDRAM.
- ***Private*-bank Assumption**: refers to whether the timing analysis of the controller assumes that the processor running the task under consideration has exclusive access to one or more banks in the SDRAM. If such assumption is made, then the task under consideration experiences no intra-bank interference, which is particularly interesting in *open-row* controllers, as the analysis can take into account

[1] In work that employs the expression page-policy instead of row-policy, the *open-row* and *close-row* policies are referred to as *close-page* and *open-page*.

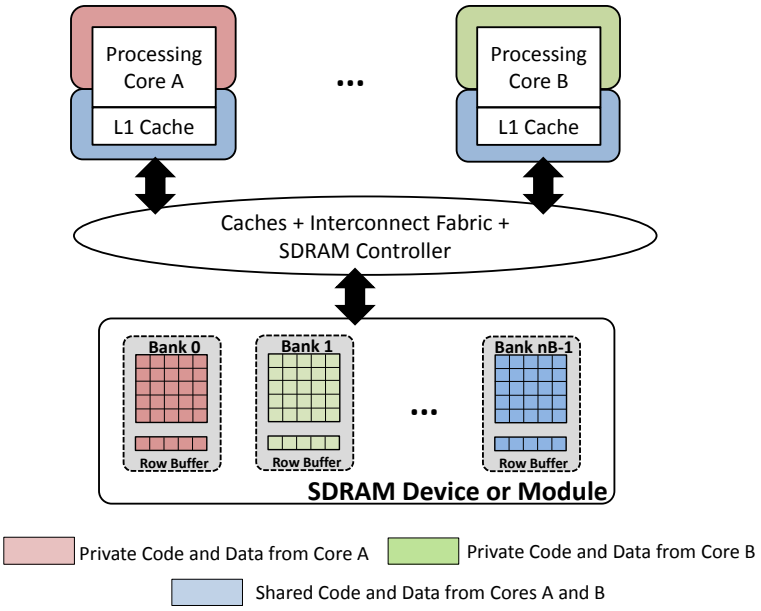


Figure 3.1.: Data sharing in scenarios in which a *private*-bank assumption is made. The background colors of the blocks representing processing cores are simply to establish a relationship between a core and the banks to which such core has access. For instance, core A can access banks 0 and nB-1.

the effect of *row buffer* locality.

In general, articles discussing controllers that employ the *private*-bank assumption overlook data sharing (e.g. [126, 77]). More specifically, they do not consider that the task under consideration must exchange data with co-executing tasks, i.e. tasks executing in other processors in the system. However, sharing can be made possible by assigning one or more banks specifically for data exchange, as depicted in Fig. 3.1. Such strategy might require (small) architectural modifications and has an influence in the timing analysis. The former is discussed in Chapter 4 and the latter in Chapter 5.

- **Refresh strategy**: refers to how the controller refreshes the SDRAM rows. For most controllers, this is achieved through the combination of a timer and of the use of the *refresh* command. More specifically, when the timer is triggered, the controller *precharges* all *row buffers* and executes a *refresh* command. Other controllers instead accomplish the task of refreshing rows by manually activating and precharging them. The advantage of using a combination of *activates* and *precharges* is that it allows a fine-grained control over the refresh operations. Namely, the *refresh* command for DDR2, DDR3 and DDR4 SDRAMs refreshes all banks simultaneously, while the manual approach can refresh banks individually. However, the manual strategy is less efficient than the *refresh* command because it refreshes fewer rows per cycle [41].

- **Mixed criticality support**: refers to whether the controller provides support for mixed criticality. This item is important because some of the controllers described in this section are purely real-time, i.e. treat all requestors in a system equally without any concern for the needs of *best-effort* applications.
- **Rank setup**: determines the number of ranks the controller expects a SDRAM module to have. Some controllers are meant for single-rank setups, some for multi-rank setups and some are designed to operate with both single- and multi-rank setups.
- **Generation-Independence**: refers to whether the approach is described in terms of abstractions of timing constraints. It is the opposite of generation-specificity, in which an approach is described in terms of constraints mentioned in the JEDEC standards for DDR SDRAMs.

The remaining of this section firstly discusses pattern-based controllers and then discusses non-pattern-based controllers. Be aware that the discussion only includes articles that propose a controller and present a corresponding timing analysis, i.e. an analytical proof of the worst-case timing behavior that the controller can display.

With regard to it, it is to be noted that before timing analyses of SDRAM controllers became widespread, Heithecker et al. [50, 51] already proposed a controller that made real-time and mixed-time-criticality considerations. More specifically, the controller supports two classes of request: high-priority (for latency sensitive traffic) and standard-priority (for throughput-sensitive traffic). The controller uses a non-*interleaved* request-to-bank mapping and serves each request with a pattern that contains an *activate* command and a CAS command with the Auto-Precharge flag, i.e. *close-row* policy. Although no formal timing-analysis is presented, the scheduling of the controller is carefully designed to prevent unbounded latencies for high-priority requests.

3.2.1. Pattern-Based Controllers

This subsection employs a simple format: firstly, the controller name is highlighted. Then, a discussion about its features is presented.

Predator et al.: The article describing the Predator controller [5] is the most widely cited article to discuss the real-time aspects of SDRAM command scheduling. With regard to it, the expression *et al.* refers to the large body of work developed upon the two basic concepts over which Predator is built: 1) treating the SDRAM as an indivisible memory unit which can process one request at a time. And 2), operating the SDRAM using statically precomputed command patterns which are scheduled during run-time as needed [2].

Fig. 3.2 depicts a high-level diagram that shows the flow of a request through Predator. The basic idea is that after entering the controller (and being buffered), multiple pending requests must be serialized by an arbiter. In [5], the arbiter employs Credit-Controller Static Priority (CCSP) [9]. In other articles, other strategies such as Frame-Based

[2]Notice that this is different than having a fully static controller, such as the one from [11].

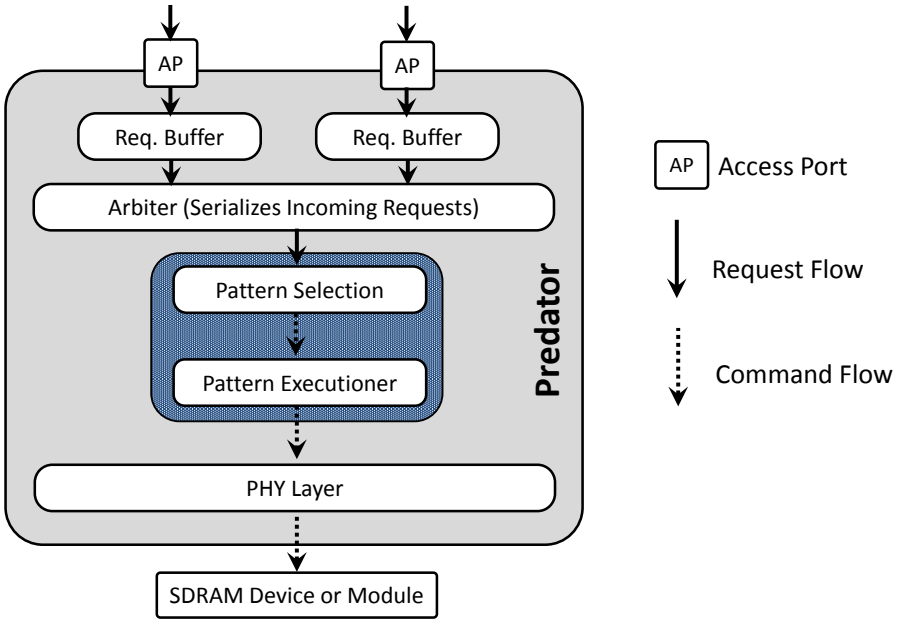


Figure 3.2.: High-level diagram depicting the flow of requests inside Predator. The data flow is omitted for the sake of clarity.

Static Priority (FBSP) [4], Priority-Based Budget Scheduling (PBS) [109, 108] and Time-Division Multiplexing (TDM) [8, 87, 118] are also investigated. In any case, the arbiter sends the request that won the arbitration to a block that selects next command pattern, which is then executed.

The article that introduced Predator mentions 3 types of command patterns[3]: one to serve read requests, one to serve write requests and one to perform *refresh* operations. Each read or write pattern is used to serve exactly one request and relies on the *close-row* policy, i.e. rows accessed to serve a request are *precharged* before the next request is processed (which is accomplished by appending CAS commands with the Auto-Precharge Flag). In [5], a request is served by executing one CAS command in each of the banks of the SDRAM device or module (i.e. one burst in each of the banks). Hence, if the controller operates a SDRAM device with $nB = 4$ banks with $W_{BUS} = 16$ using $BL = 8$, the granularity of each request amounts to $nB \cdot W_{BUS} \cdot BL = 512$ bits or 64 bytes.

In order to understand the benefits of this strategy, Fig. 3.3 depicts the state of the banks of an SDRAM device with 4 banks while a total of 3 requests are processed (two reads and one write). Notice that while data is transferred into/from one of the banks, the remaining banks are busy processing *precharge* and *activate* commands. Hence, even though the *close-row* policy is employed, the overhead for *closing* and *opening* rows is hidden, allowing the data bus to be fully utilized between the two first requests. When

[3]To be highlighted is that between the execution of a read followed by a write pattern or vice-versa, NOP commands are executed because of data bus turnarounds. In [7], these NOPs were placed into the so-called R/W or W/R patterns.

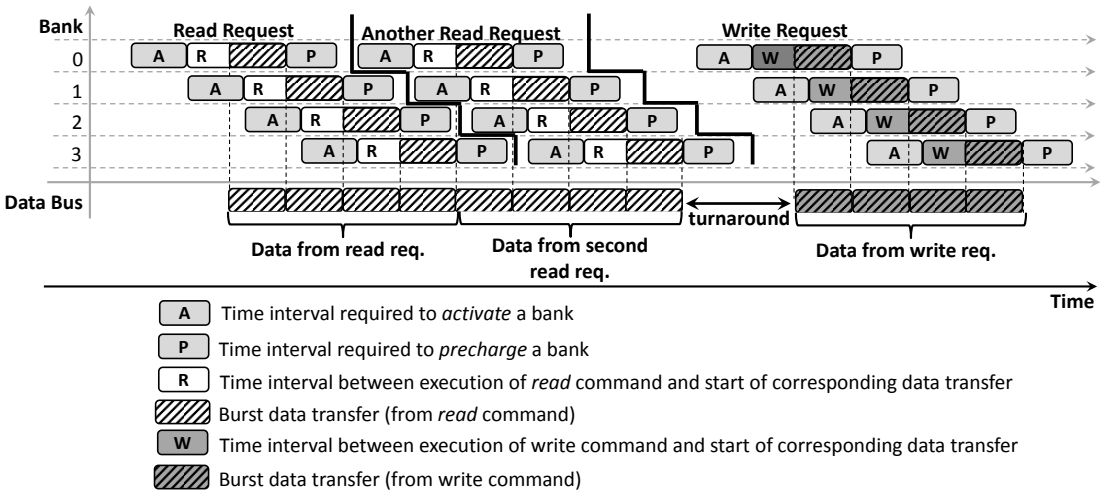


Figure 3.3.: Timing diagram depicting state of banks as command patterns are executed in a device (or module) with 4 banks. The execution of commands itself is omitted to keep the figure clear.

the third request is executed, however, there is an idle bubble in the data bus, which is a consequence of a data bus turnaround.

After the initial article, the original idea of Predator has been improved/extended in several different aspects. A comprehensive discussion about them is out of the scope of this dissertation and, hence, only an overview in form of a bullet list is provided:

- [39] discusses a more flexible approach for read and write command patterns. More specifically, it investigates patterns which access an arbitrary number of banks (instead of accessing all banks) and which can execute more than one CAS command per bank. Such patterns have implications both on request granularity and on power consumption.
- [7] proposes strategies to automatically generate SDRAM command patterns.
- [80] discusses a controller that supports multiple request granularities [80].
- [113] investigates how to exploit the run-time power down feature of SDRAMs in order to reduce power consumption. This involves carefully activating a power-saving mode from SDRAMs in times in which there are no pending requests at the memory controller.
- [30] and [32] address scheduling considerations in multi-channel setups. A SDRAM channel, in this context, refers to a logical entity comprised of a data bus, a command bus and a set of chip-select signals.
- [31], [33] and [34] cover strategies to integrate SDRAM controller and NoC.
- [40] investigates support for predictable mode changes. A mode change refers to a change in the set of active tasks or applications in a system. Mode changes have been the subject of many articles in the real-time systems community [92, 91, 114,

99, 93, 110], although [40] is (to the best of the knowledge of the author of this dissertation) the first to consider it from the perspective of a SDRAM controller.

- [37] discusses the implementation of a conservative *open-row* policy, which improves the average-latency of a request without damaging its worst-case. The intuition is that if two or more outstanding consecutive requests target the same rows of the SDRAM device or module, then there is no need to *precharge* and *activate* rows between such requests. This is accomplished by carefully redesigning the command patterns.
- [38] discusses generation-independent command scheduling.

Finally, it is important to highlight that the approach from Predator also has drawbacks, which are mostly related to two factors: 1) SDRAM technology and speed bin and 2) ratio between request granularity and data bus width. The first factor is important because there is an observable trend correlating technology/speed bin with the time interval demanded for *closing* and *opening* rows. More specifically, the faster a SDRAM device is, the larger the number of cycles required for *closing* and *opening* rows is (see Figs. 2.4a, 2.4b and 2.4c in Section 2.1.2). Hence, the *interleaved* mapping is unable to fully hide the overhead for closing and opening banks, leading to decreased data bus utilisation (imagine how Fig. 3.3 would look like if the intervals for *closing* and *opening* rows were 4x larger).

The second factor is important because it affects the possibility of employing an *interleaved* request-to-bank mapping. For instance, consider a SDRAM module with a 64-bit data bus being shared between processors that only read (or write) cache lines from (or into) it. In such scenario, assuming $BL = 8$, a single CAS command already triggers the transfer of 64 bytes (a common cache line size) and, hence, there is no need for a command pattern that accesses more than a single bank. Consequently, there is no effective way to hide the overhead for *closing* and *opening* rows.

AMC and RTCMC: The Analyzable Memory Controller (AMC) [97] (which is referred to as Real-Time Capable Memory Controller (RTCMC) in [96]) also relies on an *interleaved* request-to-bank mapping and on the *close-row* policy. In contrast to Predator, however, the AMC/RTCMC has logic that generates the commands of a pattern on-the-fly (although the pattern itself is statically designed).

The controller considers two classes of applications: Hard Real-Time (HRT), which corresponds to the *critical* level discussed in Section 1.1.1, and Non Hard Real-Time (NHRT), which corresponds to the *best-effort* level discussed in the same section. From the arbitration perspective, requests from HRT applications have priority over the ones from NHRT. Moreover, the controller includes logic to preempt Soft Real-Time (SRT) requests in case a HRT arrives. Inside the same level of criticality, however, a round-robin schedule is employed and no preemptions are performed.

In summary, the controller has similar advantages and drawbacks as Predator.

PRET: The PREcision Timed controller (PRET) controller [100] also employs the *close-row* policy. With regard to Predator, PRET brings three innovations:

- The use of a non-*interleaved* request-to-bank mapping and of a *private*-bank assumption. More specifically, unlike Predator, which treats the entire SDRAM as an indivisible resource, PRET treats each set of 2 banks as an *independent* resource. As [100] considers SDRAM modules with a total of 8 banks divided into 2 ranks, PRET has a total of 4 *independent* memory resources.

  Notice that the word *independent* is written in italic. That is because although the banks of a SDRAM module enjoy a certain degree of independence from each other, they share command and data buses. In order to enforce that each *independent* memory resource gets its fair share of command and data bus time, a schedule with TDM slots is employed. Each time slot is used to serve a single request and includes an *activate* command and a single CAS command with the Auto-Precharge flag.
- The use of rank interleaving. Namely, consecutive time slots are always granted to *independent* memory resources that are located in different ranks. Hence, the overhead for data bus turnarounds can be potentially hidden. Nevertheless, it is important to observe that [100] considers slow DDR2 devices, in which the overhead for turnarounds is small. As SDRAMs get faster, such overhead gets larger and can no longer be covered with a dual-rank setup. Moreover, the rank switching overhead also gets more significant.
- The use of manual refreshes (instead of relying on the *refresh* command). More specifically, a row can be refreshed by issuing an *activate* and a *precharge* command to it. As discussed in the beginning of Section 3.2, this allows the controller to have a fine-grained control of the refresh operation at the cost of efficiency.

Last, it is also important to mention that the controller has been integrated into a so-called PRET machine [28, 81, 82], an architecture that provides strict timing isolation (i.e. *composable* service discussed in Section 1.1.1) for multiple co-running applications. However, neither the controller nor the target PRET machine provide support for mixed criticality.

MCMC and Throughput-Aware[4] MCMC:

Both the Mixed Critical Memory Controller (MCMC) [27] and the Throughput-OrientedMCMC [26] were proposed by the author of this dissertation and are based upon the concept of *independent* memory devices employed by PRET.

In summary, the main contributions of [27, 26] are:

- With regard to the TDM rank-interleaved schedule employed by PRET, [27] discusses different bank partitioning schemes and different command patterns inside each time slot. So, for instance, [27] shows that it is possible to group the 4 *independent* memory resources (mentioned in the PRET controller article) into pairs, thus

[4]The expression Throughput-Oriented is not employed in [26] and was coined solely for the sake of the discussion.

forming two larger resources. From the TDM schedule perspective, the time slot for each larger resource is composed using the two time slots from the original *independent* memory resources. This means that each time slot from a larger resource will access two different banks (one for each of the original time slots). Notice that such strategy modifies the baseline request-granularity of the controller. Moreover, it represents a change from a non-*interleaved* into an *interleaved* request-to-bank mapping.

- The MCMC [27] adds support to mixed criticality. More specifically, it includes two request queues for each *independent* memory resource: one for a (single) *critical* requestor and one for (one or more) *best-effort* requestors. Pending requests from the *critical* requestor have priority over *best-effort* ones. Requests in the *best-effort* queues are only served if the corresponding *critical* queue is empty. Combined with the TDM arbitration employed to alternate between the *independent* memory resources, each *critical* requestor in the system is completely isolated from the timing perspective.
- The Throughput-Oriented MCMC [26] also considers mixed criticality but addresses a scenario in which the *critical* requestors demand a throughput guarantee (instead of a fine-grained guarantee). In such cases, if fixed priority is used to arbitrate between *critical* and *best-effort* within an *independent* memory resource, a burst of requests from the former can unnecessarily block requests from the latter. Hence, [26] prioritizes *best-effort* requests over *critical* while at the same time throttling *best-effort* traffic using a traffic shaper. This is similar to having a CCSP arbiter [9] with two priority levels and with the highest-priority (in this case the *best-effort*) being credit-controlled. Such strategy reduces the average latency of *best-effort* requests while still allowing throughput guarantees to be computed for *critical* requestors.

The ideas of both the MCMC and the Throughput-Oriented MCMC suffer from the same drawbacks as PRET, i.e. lack of scalability.

PMC: The Programmable Memory Controller (PMC) (initially discussed in [48] and extended in [49]) targets mixed criticality environments and supports different request sizes. The controller relies on precomputed command bundles that resemble the patterns employed in the article that introduced the conservative *open-row* policy [37].

However, the authors from [48, 49] use the expression *mixed-row* to refer to their *row buffer* policy. In comparison with the conservative *open-row* policy from [37], the following differences are highlighted:

- The *mixed-row* policy exploits locality within the boundary of a single large request (which can be served with one or more command bundles, not necessarily executed consecutively). The conservative *open-row* policy, however, attempts to exploit SDRAM locality over the boundary of a single request.
- The *mixed-row* policy is designed to improve both the average and the worst-case (the improvement in worst-case is achieved by executing two or more bundles of

the same request consecutively). The conservative *open-row* has as goal improving the average case, while keeping the worst-case equal to the one observed with the *close-row* policy.

- In the *mixed-row* from PMC, the exploitation of *row buffer* locality depends on how many command bundles for a request are allowed to execute consecutively. In the conservative *open-row* policy, a decision to keep a row open depends on the arrival time of consecutive requests, i.e. a row is left open only for a predetermined time window.

In order to define which command bundle to execute next, the PMC relies on a TDM schedule, which is statically precomputed by taking into account the latency and throughput demands of all requestors in the system. Such schedule is loaded into the controller during *boot-time*, hence the *programmable*[5] in the name.

In summary, as it was the case for Predator, the controller is highly effective as long as the ratio between request granularity and data bus width is large.

3.2.2. Non-Pattern-Based Controllers

Non-pattern-based controllers appeared to handle scenarios in which the granularity of the requests performed by processors is not significantly larger than the data bus width of the SDRAM module employed by the system. For instance, in a SDRAM module with a 64-bit data bus operated with $BL = 8$, a single CAS command transfers 64 bytes, which is large enough in a system in which processors only make cache-line sized requests. In such cases, it is consequently not possible to exploit the *state* of the SDRAM within the boundary of a single-request. More specifically, a controller must exploit the *state* of the SDRAM over the boundary of different requests (not necessarily issued by the same requestor).

The remaining of the discussion about non-pattern-based controllers follows the same format employed in Section 3.2.1.

ORP: The Open-Row Private bank controller (ORP)[6] [126, 127] is the first controller in the literature to employ the *open-row* policy in the real-time domain. The main idea of the approach is to use the *open-row* policy and a non-*interleaved* request-to-bank mapping with the *private*-bank assumption, i.e. giving a requestor exclusive access to one or more banks of the SDRAM. The intuition is that consecutive requests issued by the same application will potentially target the same row in the memory because of the principle of spatial locality [7].

The article that introduced ORP focuses mainly on a timing analysis of the approach, and not on the logical structure of the hardware that implements it. However, for ease of understanding, a possible architecture that implements the approach is depicted in Fig. 3.4. In summary, the controller is requestor-oriented, i.e. its operation depends

[5] Notice that this is different than reconfiguration at *run-time* discussed in [40].

[6] The acronym ORP was coined in [42] and was not employed originally in [126].

[7] Spatial locality is one of the principles that guides the design of cache memories (the other being temporal locality) [35, 36].

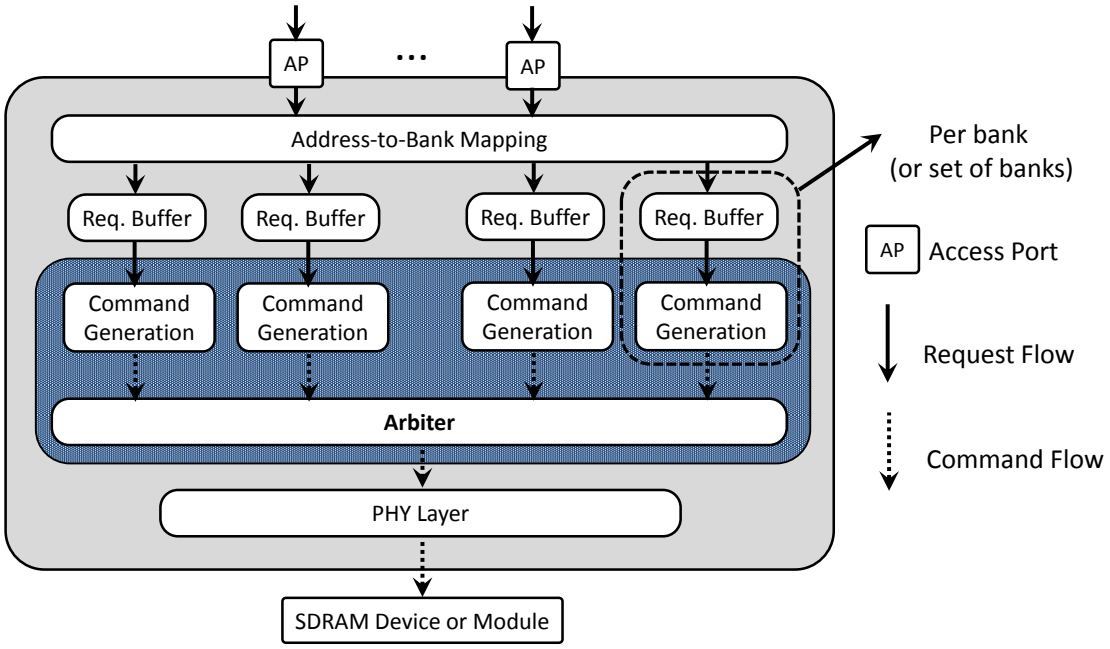


Figure 3.4.: High-level diagram depicting the logical structure of a non-pattern-based controller. The data flow is omitted for the sake of clarity.

on the number of requestors in the system (e.g. 4 cores equals 4 requestors). Each requestor is assigned exclusive access to a set of one or more SDRAM banks and has its own request buffer and command generation block.

The command generators are responsible for translating a request into the proper set of SDRAM commands that will serve such request. With that regard, it should be noted that the controller supports a single request granularity, which is equal to $BL \cdot W_{BUS}$ bits. This means that each request demands exactly one CAS command and, depending on whether a new row needs to be loaded into the corresponding *row buffer*, an *activate* and a *precharge* commands. Hence, incoming requests can be classified into 4 types, which are enumerated in Table 3.1. In the table, the word *miss* refers to the *row buffer* of the SDRAM (and not to a cache memory).

Table 3.1.: Types of request in ORP.

Type	Command Sequence	Mnemonic
Read Miss	P-A-R	RM
Read Hit	R	RH
Write Miss	P-A-W	WM
Write Hit	W	WH

The command generators forward commands into an arbiter, which selects among all pending commands the next command to be executed. It is very important to notice that the arbitration stage handles commands which belong to requests made by different requestors. This is different from what is observed in a pattern-based controller, in which all commands necessary to serve a request are executed before any commands necessary to serve the following request. To clarify the matter, the interested reader should compare Fig. 3.4 with Fig. 3.2.

The command arbiter is now discussed. As already mentioned, the articles that introduce the ORP focus on the real-time properties of the controller, and not on its implementation. Hence, the operation of the command arbiter is described using a set of rules. More specifically, the arbiter contains a global command queue, which is operated using four rules:

1. Each requestor can have at most one pending command inserted into the global queue.
2. A command can only be inserted into the global queue if all timing constraints related to previous commands generated by the same command generator are satisfied.
3. The arbiter prioritizes the oldest command in the queue that can be immediately executed without violating a timing constraint. An exception is made for CAS commands and is discussed in the fourth rule.
4. If the oldest CAS command in the queue cannot be immediately executed without violating a constraint, any other CAS command in the queue is ignored.

Notice that the third rule implies that an older command is not always executed before more recent commands. This is interesting because it increases the overall performance of the controller, as can be seen in in Fig. 3.5. In the figure, the command queue is depicted in the left side, while the execution of commands performed by the arbiter is depicted in the right side. Notice that the *precharges* for banks 2 and 3 are executed before the *activate* in bank 1. That is because after the execution of the *activate* for bank 0, at least $dAA\text{-}R\overline{B}$ cycles must pass before another *activate* can be executed, while *precharges* can be executed immediately.

Notice also that the fourth rule enforces that an older CAS command is always executed before a more recent CAS command. Such rule (which is an exception to third rule) is designed to prevent an unbounded latency for CAS commands. In order to understand the reason, consider the scenario from from Fig. 3.6.

In the figure, the black circle in bank 2 (instant t_1) marks the moment in which the *write* from bank 2 could be executed without violating any timing constraint. Hence, if the controller did not employ the fourth rule, it would execute the *write* from bank 2 in t_1, thus postponing the *read* from bank 1. Such execution would represent a straightforward application of the third rule, given that it is only possible to execute the *read* from bank 1 in t_2. From the real-time perspective, this would mean that the *read* under consideration could be indefinitely postponed by a successive stream of *write* commands,

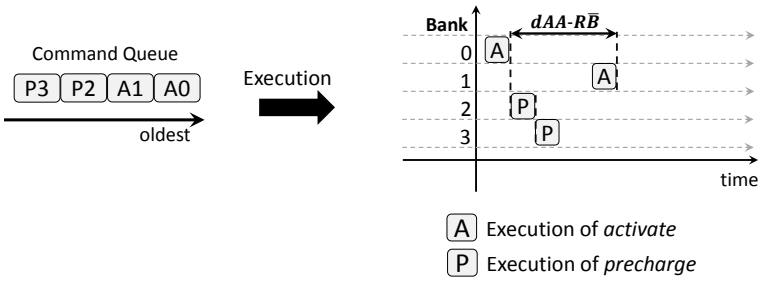


Figure 3.5.: Example of scheduling of *activates* and *precharges* performed by the arbiter (third operational rule). The numbers inside the command queue represent the target bank of a command, e.g. P3 represents a *precharge* to bank 3.

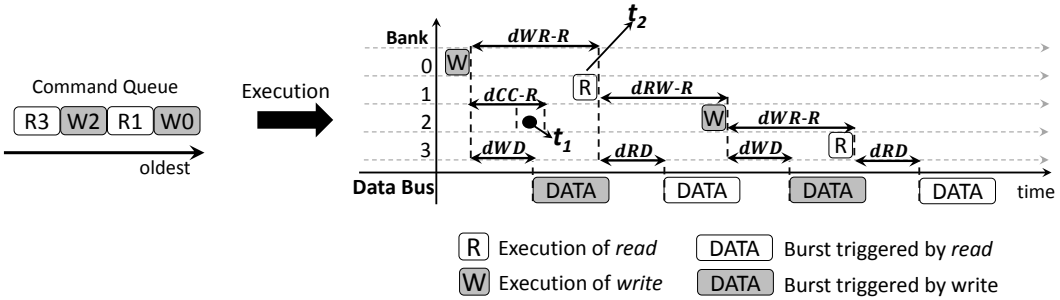


Figure 3.6.: Example of scheduling of CAS commands performed by the arbiter.

i.e. unbounded latency.

Two further considerations are made about the fourth rule: 1) the didactic example (on the implications of not having the fourth rule) considered a *read* command being blocked by *write* commands, but the opposite is also possible. 2) Although necessary to enforce a bounded latency on CAS commands, the fourth rule has two drawbacks: it decreases data bus utilisation and forces designers to assume an alternating pattern of *reads* and *writes* when computing timing guarantees. Both issues are tackled by the controller proposed in this dissertation.

The timing analysis is now discussed. Unlike pattern-based controllers, which give latency guarantees in terms of individual requests, the timing analysis from [126, 127] computes a guarantee in terms of all requests performed by a task. More specifically, it computes a worst-case cumulative SDRAM latency, which refers to the maximum amount of time a task spends idle waiting for its SDRAM requests to be served. For ease of understanding, Fig, 3.7 depicts a simplified diagram of how the execution of a task is divided between computation and memory time. The timing analysis from [126, 127] computes an upper-bound on the gray area.

The motivation for giving guarantees over all requests is to be able to capture the effect of *row buffer* locality. Namely, in order to compute guarantees for an individual request, a conservative bound must assume that such request misses at the *row buffer*,

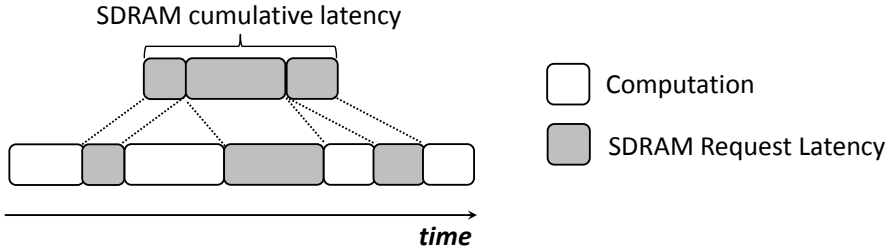


Figure 3.7.: Cumulative SDRAM latency.

which leads to larger latency as its demands an *activate* and a *precharge* (in addition to a CAS). Hence, the authors from ORP mention that a static timing analysis tool [15] can be employed to extract the number and type of requests that an application can make[8] (see Table 3.1). This allows the computation of the worst-case bound to take into account that not all requests demand *precharges* and *activates*.

Finally, is is important to observe that [126, 127] do not explicitly mention mixed criticality. However, nothing prevents a designer from assigning banks for *best-effort* requestors and modifying the corresponding bank schedulers to employ the FR-FCFS policy.

ROC: The Rank-Switching Open-Row Controller (ROC)[77] bears the following similarities to the ORP: it is also requestor-oriented; it supports a single request granularity; and it employs a combination of *open-row* policy, non-*interleaved* request-to-bank mapping and a *private*-bank assumption (notice that Table 3.1 also applies to ROC).

However, in comparison with ORP, two main differences are highlighted: firstly, ROC is a controller for multi-rank modules. More specifically, as it will become clear, it is a non-pattern-based controller that uses the multi-rank structure to mitigate the overhead for data bus turnarounds (notice that controllers such as PRET and MCMC also do so, but rely on command patterns and on the *close-row* policy).

And secondly, the article that introduced ROC explicitly mentions mixed criticality. Namely, if the controller operates a module with nR ranks, H of them are assigned to Hard Real-Time (HRT) requestors (corresponds to *critical* in 1.1.1) and S of them are assigned to Soft Real-Time (SRT) requestors (corresponds to *best-effort* in Section 1.1.1) as long as $H + S = nR$. Intra-rank command scheduling in SRT ranks is left mostly up to the designer (which can employ performance-oriented optimizations without worrying about real-time considerations). Sufficient timing isolation for requestors in HRT ranks is enforced using a strict set of rules for inter-rank scheduling.

From the implementation perspective, ROC can employ a structure similar to the one depicted in Fig. 3.4, as long as each HRT requestor is assigned its own request buffer, code generation block and a set of one or more banks inside one of the ranks of the system. SRT requestors can (although they do not necessarily need to) share buffer and

[8] Another possibility would be to use measurement-based techniques [2, 70, 121, 17, 1, 21]

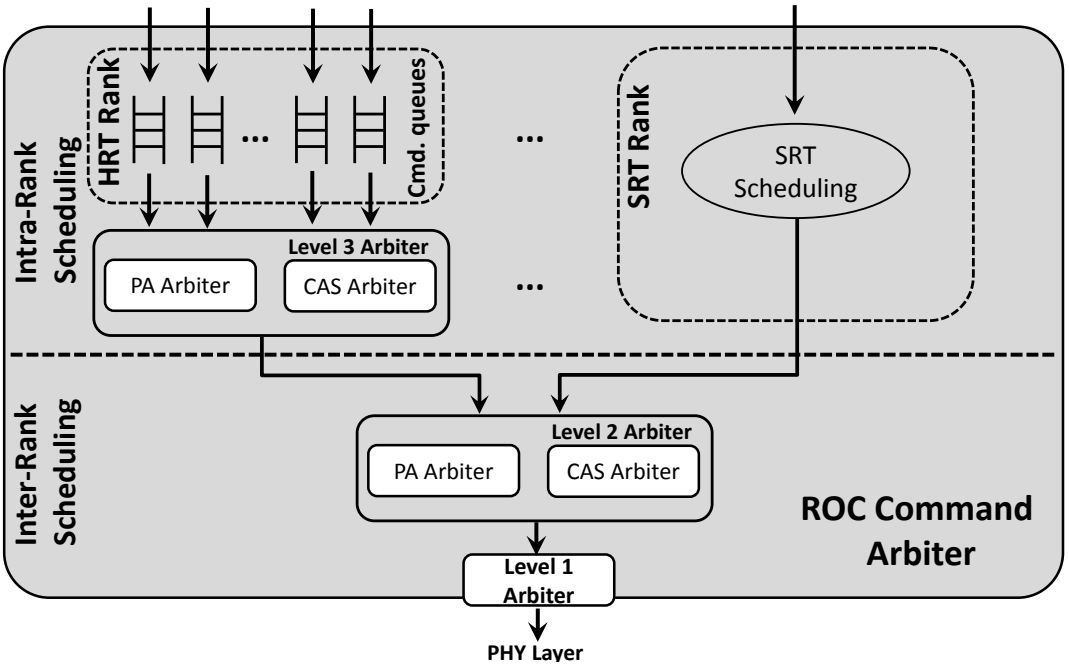


Figure 3.8.: ROC command arbiter.

command generators.

The (command) arbiter design is described with a block diagram and a set of scheduling rules. The block diagram is depicted in Fig. 3.8. From the figure, the following should be noticed:

- The arbiter has a multi-level organization. The idea is that each level implements a predetermined set of rules (discussed later) and propagates the winning command into the next layer. Moreover, levels 2 and 3 refer to inter-rank scheduling, while anything before level 2 refers to intra-rank scheduling.
- The Level-3 Arbiter of HRT ranks and the Level 2 Arbiters (regardless of rank type) are subdivided into a PA Arbiter (which handles *precharge* and *activate* commands) and a CAS Arbiter, which handles *read* and *write* commands.
- Each requestor assigned to a HRT rank has its own command queue within the arbiter. Moreover, the intra-rank scheduling performed for a HRT rank (Level-3 Arbiter) is guided by an specific set of rules (discussed later).
- The intra-rank scheduling of SRT ranks is left mostly to the designer. In the figure, such degree of flexibility is represented by the *SRT Scheduling* block.

That being said, the rules that dictate the operation of the arbiter are now enumerated:

1. The command at the head of each per-requestor queue is said to be *active* if all timing constraints that are caused by previous commands of the same requestor are satisfied. Moreover, a CAS command does not become *active* until data the

data of the previous CAS command of the same requestor has been transmitted.

2. The L3 PA Arbiter uses a modified First-Come First-Served (FCFS) arbitration. More specifically, the oldest *active activate* or *precharge* is propagated to L2 PA.
3. The L3 CAS Arbiter uses simple FCFS (i.e. even CAS commands that cannot be immediately executed can be selected, unlike the case for *activates*). Moreover, it also computes the earliest time at which the data transmission for the corresponding requestor could be started, which is referred to as t_{SD_r}. For that purpose, it considers the last CAS command executed by the controller (regardless for which rank). Both the selected CAS command (using FCFS) and the corresponding t_{SD_r} are forwarded into the next level of arbitration. (Notice that rule 3 differs from rule 2 in the sense that even CAS commands that cannot be immediately executed can be forwarded into the next level).
4. The L2 PA Arbiter can employ either FCFS or Round-Robin (RR).
5. The L2 CAS Arbiter uses a modified FCFS arbitration that allows a CAS command from a rank constrained by dRW-R or dWR-R (data bus turnarounds) to be preempted by CAS commands in ranks which are not constrained by such constraints. Formally speaking, let t_{ED} be the time at which the data transmission of the last issued CAS command ends. The L2 CAS Arbiter will select the oldest pending CAS command for which it holds that $t_{SD_r} \leq t_{ED} + t_{RTRS}$.
6. The L1 Arbiter simply prioritizes CAS commands over *activates* and *precharges*[9].

The name of the controller is derived from the fact that the controller employs rank-switching in order to mitigate the overhead for data bus turnarounds. An example of how the controller operates is depicted in 3.10. In the figure, the idle periods in the data bus between consecutive bursts amount to t_{RTRS}, which is smaller than the idle periods observed in ORP in a similar scenario (see Fig. 3.6). Notice, however, that Fig. 3.6 considers a single-rank scenario, while Fig. 3.9 considers a four-rank scenario.

It is also interesting to observe that Rule 5 allows the rank-switching logic to be bypassed in case a rank is constrained by data bus turnarounds. An example of such case is depicted in Fig. 3.10. In the figure, the second *read* for rank 2 is executed before the *read* for rank 0, even though the latter is older than the former. This is because rank 0 is constrained by dWR-R.

ROC has, however, three main drawbacks: firstly, the larger the overhead for data bus turnarounds, the larger the number of ranks ROC needs to be effective. This is specially important because, as discussed in Section 2.1.3, the overhead for data bus turnarounds is steadily increasing for newer devices.

Secondly, extra ranks are not cheap. More specifically, assuming the same SDRAM technology, a dual-rank module has two times more chips than its single-rank counterpart. This has implications both from the product cost and from the power consumption perspectives.

[9] In [77], Rule 6 is not explicitly enumerated. However, the first paragraph of the second column of page 4 from [77] mentions that the L1 level prioritizes CAS over *activates* and *precharges*.

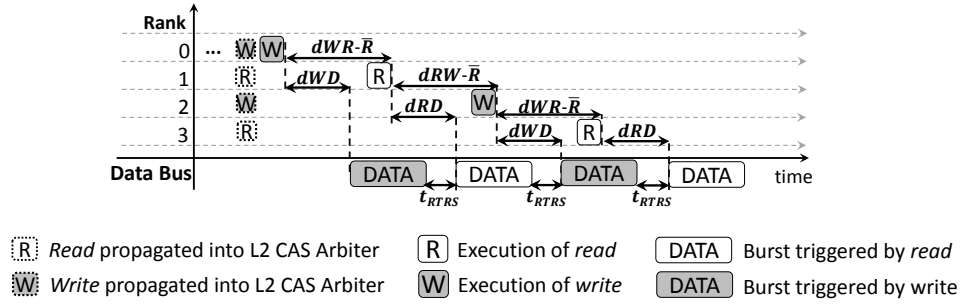


Figure 3.9.: Example of scheduling performed by ROC. Notice that the y-axis refers to the pending commands of ranks (which are provided by the Level 2 CAS Arbiters). This is **different** than the y-axis from Figs. 3.5 and 3.6, which represents banks.

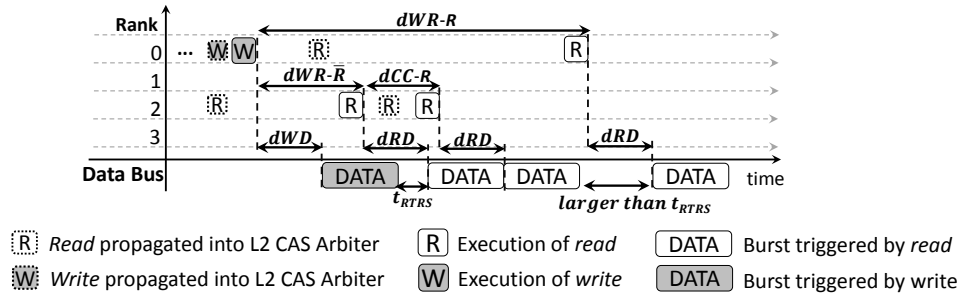


Figure 3.10.: Another example of scheduling performed by ROC. The $dWR\text{-}R$ constraint is deliberately depicted as very large to highlight the *modified-nature* of the FCFS arbitration employed by the Level 1 Arbiter.

And finally, although the rank-switching overhead (t_{RTRS}) is smaller than the overhead for data bus turnarounds (see Sections 2.1.3 and 2.2), it is still able to cause a drop in the utilisation of the data bus by more than half, depending on the operating frequency of the SDRAM module.

DCmc: The Dual-Criticality memory controller (DCmc) [59] is a single-rank *open-row* controller that supports a single request granularity and uses a non-*interleaved* request-to-bank mapping. The name of the controller refers to the two classes of criticality that are identified as important for applications in the space domain: real-time, which corresponds to the *critical* level described in Section 1.1.1, and high-performance, which corresponds to the *best-effort* level in Section 1.1.1.

The main idea of the approach is to partition the banks of a SDRAM module between the two criticality levels. The exact number of banks that each partition has can be configured by the operating system. Inside each partition, requestors belonging to the same criticality level can compete for the same bank in the SDRAM, i.e. no *private*-bank assumption is made. As a consequence, guarantees for real-time requestors must assume

that each request misses at the *row buffer*.

The article that introduces DCmc [59] makes a clear distinction between intra- and inter-bank arbitration. Intra-bank arbitration determines which request is going to be served next, given a set of requests that target the same SDRAM bank. The intra-bank arbitration can be summarized with two rules:

- Inside a bank from the real-time partition, round-robin is employed.
- Inside a bank from the high-performance partition, FR-FCFS is employed.

After the selection is made, the intra-bank arbiter generates the appropriate set of SDRAM commands to serve the request, which are then sent into the inter-bank arbiter. Given the set of pending commands from all intra-bank arbiters in the system, the inter-bank arbiter decides the next command to be executed. The operation of the inter-bank arbiter can be summarized with two main rules:

- Commands from requests of the real-time level have priority over commands of the high-performance level.
- Inside each criticality level, pending commands are arbitrated using round-robin.

As it is the case for the ORP, no optimization to mitigate the impact of data bus turnarounds is made.

CMD-Priority: CMD-Priority[10] [66] is a predictable and command-level priority-based SDRAM controller for mixed criticality systems. The criticality levels supported by the controller are similar to the ones considered in this dissertation (see Section 1.1.1), i.e. *critical* and *best-effort*.

The CMD-Priority is a single-rank *open-row* controller that supports a single request granularity and uses a non-*interleaved* request-to-bank mapping. Its biggest difference with regard to the other non-pattern-based controllers discussed in this section is that it allows requestors from different criticalities to share banks of the SDRAM. More specifically, each bank can be shared by up-to-one *critical* requestor and multiple *best-effort* requestors. The up-to-one restriction is to ensure that different *critical* requestors only compete for a bank with *best-effort* requestors. Combined with prioritization of commands from *critical* requests over the ones from their *best-effort* counterparts, each *critical* requestor is sufficiently isolated (from the timing perspective) from other requestors in the system.

From the command scheduling perspective, the operation of CMD-Priority can also be described in terms of intra- and inter-bank arbiters. The operation of the intra-bank arbiters are guided by the following rules:

- A *critical* request always has priority over *best-effort* requests.
- *Best-effort* requests are prioritized using FR-FCFS.
- If a *critical* request arrives while a *best-effort* request is being served, the *best-*

[10]The authors of [66] did not explicitly assign a name to the controller. To refer to it, this dissertation uses the term CMD-Priority, which was coined in [41].

Table 3.2.: Summary of Pattern-Based SDRAM Controllers

Controller	Row-Policy	Mixed-Criticality	Request-to-Bank Mapping	*Private*-bank Assumption	Refresh	Rank-Setup	Generation-Independence
Predator et al.	*close-row* and cons. *open-row*	Yes	*interleaved*	No	timer + *refresh*	Single-Rank	Yes (see [38])
AMC/RTCMC	*close-row*	Yes	*interleaved*	No	timer + *refresh*	Single-Rank	No
PRET	*close-row*	No	non-*interleaved*	Yes	manual	Multi-Rank	No
MCMC	*close-row*	Yes	setup-dependent	No	manual	Multi-Rank	No
PMC	*mixed-row*	Yes	*interleaved*	No	refers to [67]	Single- or Multi-Rank	No No

Table 3.3.: Summary of Non-Pattern-Based SDRAM Controllers

Controller	Page Policy	Mixed Criticality	Request-to-Bank Mapping	*Private*-bank Assumption	Refresh	Rank Setup	Generation-Independence
ORP	*open-row*	No	non-*interleaved*	Yes	timer + *refresh*	Single-Rank	No
ROC	*open-row*	Yes	non-*interleaved*	Yes	timer + *refresh*	Multi-Rank	No
DCmc	*open-row*	Yes	non-*interleaved*	No	refers to [12]	Single-Rank	No
CMD-Priority	*open-row*	Yes	non-*interleaved*	No	timer + *refresh*	Single-Rank	No
RW-Bundler (this dissertation)	*open-row*	Yes	non-*interleaved*	Yes	refers to [126]	Single-Rank or Multi-Rank	Yes

effort request is preempted. This demands command-level preemption logic, which generates the required commands to prepare the bank under consideration for the *critical* request.

The operation of the inter-bank arbiter follows the same two principles as the one from the DCmc. Moreover, the CMD-Priority bears two other similarities to the DCmc: firstly, guarantees for the *critical* requestors are computed assuming that their requests always miss at the corresponding *row buffers*. And secondly, no optimization to mitigate the impact of data bus turnarounds is made.

3.3. Summary and Distinguishing Features of This Work

This chapter presents the related work on SDRAM controllers. Firstly, it briefly discusses controllers that are average-performance-oriented. More specifically, the chapter clarifies that such controllers employ optimizations that lead to hard-to-predict behavior and, hence, are not suited for real-time and mixed criticality domains.

Controllers for the real-time and mixed criticality domains are presented in the second part of this chapter. In summary, they are divided into main groups: pattern-based, that relies on statically precomputed command patterns, and non-pattern-based, that generates commands dynamically. The former excels in scenarios in which the ratio between request granularity and data bus width is large. The latter is better suited if such ratio is small.

Within the same group, controllers can be further classified using the following criteria: row-policy, support (or lack of support) of mixed criticality, request-to-bank mapping, *private*-bank assumption, refresh mechanism, rank setup and generation-independence. Using such criteria, the features of pattern-based and non-pattern-based controllers are

summarized in Tables 3.2 and 3.3, respectively. The latter also describes the features of the controller proposed in this dissertation (RW-Bundler).

With regard to its non-pattern-based counterparts, the *RW-Bundler* performs aggressive reordering of CAS commands in order to minimize the number of data bus turnarounds and rank switching events. As it will become clear, the reordering of commands is carefully designed in order to not only improve the timing bounds for *critical* requestors but also to improve data bus utilisation (and, hence, the performance of *best-effort* requestors). Moreover, the RW-Bundler is generation-independent and can be configured to operate with an arbitrary number of ranks.

4. An SDRAM Controller Architecture for Mixed Criticality Systems

This chapter proposes a multi-generation *open-row* SDRAM controller architecture that supports arbitrary SDRAM module configurations, i.e. an arbitrary number of ranks (nR), number of *bank groups* per rank (nG)[1] and number of banks per rank (nB). The architecture performs SDRAM command scheduling based on minimum distances between consecutive commands, which are discussed in Section 2.3. Moreover, it implements command-level read/write bundling in order to minimize the number of data bus turnarounds and rank switching events, which are pinpointed as challenges of growing relevance for mixed criticality in Chapter 2.

The controller proposed in this chapter falls into the non-pattern based category (see Section 3.2.2) and as such is better suited for scenarios in which the ratio between request granularity and data bus width is small. Moreover, it employs a non-*interleaved* request-to-bank mapping and supports a single request-granularity, which is equal to $W_{BUS} \cdot 2 \cdot t_{BURST}$, i.e. the amount of data transferred due to the execution of a single CAS command. Hence, in systems with $W_{BUS} = 64$ and $t_{BURST} = 4$, each request transfers 64 bytes, a common cache-line size. Larger requests must be chopped up in several baseline-granularity-sized requests.

Before the start of this discussion about the architecture, an **important remark** is made: the timing analyses presented in the articles that originally discussed read/write bundling [22, 25] relied on a subjective *not-too-late* assumption, which was then pinpointed in [24]. Instead of simply adjusting the timing analysis in order to compute timing bounds independently of such assumption (which worsens the timing bounds), this chapter includes a scheduling modification to handle *too-late* commands (which was proposed in [24]). Furthermore, the next chapter discusses the implications of such modification in the worst-case guarantees for real-time tasks.

The remaining of this chapter is structured as follows: Section 4.1 presents an overview of the controller architecture. Sections 4.2 and 4.3 discuss blocks that implement command generation and scheduling. Finally, Section 4.4 discusses the implications of requestor-to-SDRAM-bank-assignment and is followed by a summary of the chapter in Section 4.5.

[1] To keep the architecture generic, DDR2 and DDR3 memories are considered to have a single group ($nG = 1$) which comprises all banks of the system.

4.1. SDRAM Controller Architectural Overview

The architecture of the proposed SDRAM controller is depicted in Fig. 4.1. In the figure, nR refers to the number of ranks in the controlled SDRAM module and nB refers to the number of banks in each rank. The *bank groups* feature present in DDR4 is omitted for the sake of clarity, but is addressed in the text in the discussion about the channel scheduler (Section 4.3).

Notice that single-rank modules are also addressed by the architecture, given that they can be seen as a multi-rank module in which $nR = 1$. Notice also that, for all nB banks of each rank, there is a bank request queue, a bank scheduler and a command register. The other 3 blocks, i.e. address mapping, data buffers and channel scheduler, are shared by all $nR \cdot nB$ banks in the system.

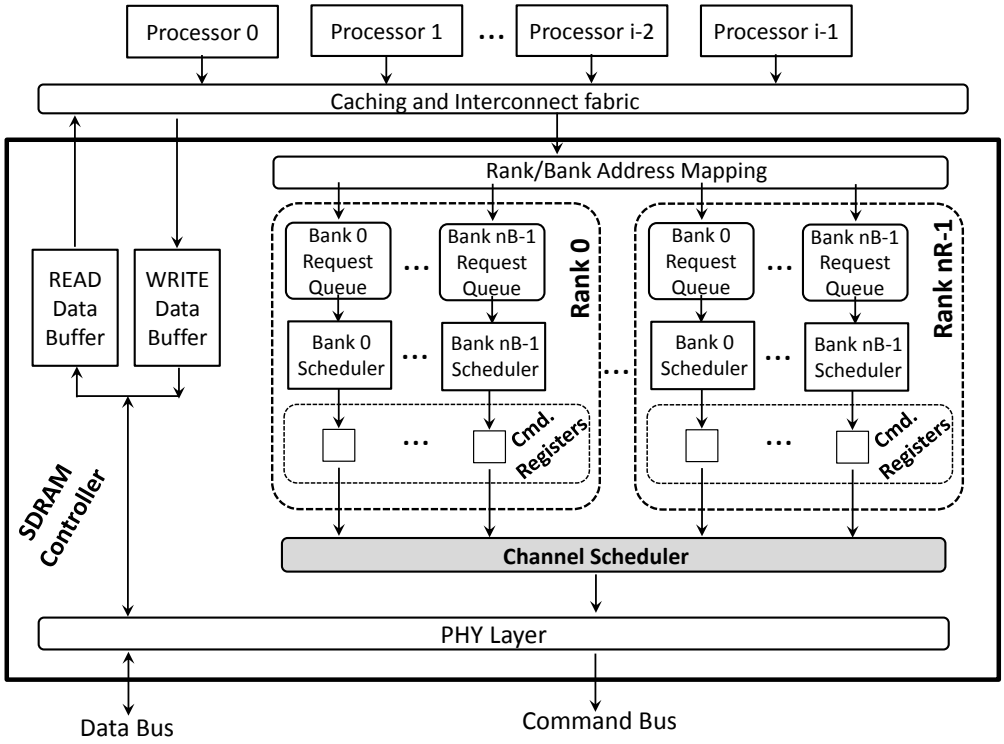


Figure 4.1.: Logical architecture of the SDRAM controller.

The flow of a request through the controller is now discussed. At arrival, incoming requests go firstly through the address mapping block [2], which decodes their addresses and forwards them to the proper bank request queue. The request queue then forwards one of its pending requests into the corresponding bank scheduler (provided that such bank scheduler already finished processing the previous request forwarded into it). The

[2]Notice that a *private*-bank setup can be implemented by carefully assigning addresses to requestors [85].

prioritization scheme employed by the request queue depends on the criticality[3] assigned to the bank: *best-effort* banks employ FR-FCFS (see Section 3.1) and *critical* banks employ FCFS. Notice that if a bank is shared between *critical* and *best-effort* requestors (see Section 4.4), then it must be designated as *critical* and will employ FCFS.

Once in a bank scheduler, the request is decoded into the commands required to serve it. Such commands are then placed into the corresponding command register. Each bank scheduler has its own command register and each command register stores exactly one command (the oldest outstanding command).

Then, the channel scheduler arbitrates between the command registers of all banks in the system and forwards the winning command into the PHY layer, which executes it. Moreover, after the data transfer triggered by the request is performed, the controller acknowledges the request completion to the corresponding requestor. In case of read requests, the acknowledgment also contains the data read from the SDRAM.

From the mixed criticality perspective, it is **very important** to notice that the distinction between a *critical* and a *best-effort* requestor happens at the intra-bank level. More specifically, *best-effort* banks employ aggressive request reordering in order to increase the exploitation of *row buffer* locality, i.e. the FR-FCFS policy. *Critical* banks stick to FCFS because of its predictable timing. However, once a request is translated into commands, such commands compete equally with all other pending commands in the system (regardless of the criticality).

The remaining of this chapter firstly discusses the bank schedulers and the command registers. Then, it provides a thorough description of the channel scheduler, including hardware synthesis results. Finally, it discusses the assignment of cores and applications to SDRAM banks. Notice that such assignment is mostly independent of the design of the controller, but is addressed in this chapter because it is relevant for the timing and safety of *critical* applications. Notice also that the design of the PHY layer is orthogonal to command scheduling and, hence, plays no role in mixed criticality. Therefore, this chapter does not further discuss it.

4.2. Bank Schedulers and Command Registers

The function of a bank scheduler is to translate a memory request into a set of SDRAM commands that fulfill such request. If the bank scheduler employs the *open-row* policy, a request is translated into either a CAS command or into a *precharge-activate-CAS* sequence, depending on whether it hits or misses at the row buffer. Notice that the proposed architecture could also implement the *close-row* policy, in which case the bank schedulers would serve requests with an *activate-*(*CAS* with Auto-Precharge) command sequence. Nevertheless, the *close-row* policy is not exploited in this dissertation as it increases power consumption.

In addition to generating the commands required to serve a request, the bank schedulers must also synchronize their operations with the corresponding command registers.

[3] Assigning a criticality to a bank can be either a design-time parameter or implemented with a configuration register.

The function of command registers is to serve as an intermediate level of buffering that decouples the implementation of the channel scheduler from the bank schedulers. There is one command register for each bank scheduler. The channel scheduler removes commands from the registers when the commands are executed (sent to the SDRAM module). This allows the bank scheduler whose register was emptied to insert a new command (after the pertinent constraints no longer pose a violation), and so on.

A bank scheduler must only place a command in its register if such command can be immediately executed (in the cycle after the insertion) by the channel scheduler without violating any *exclusively intra-bank* timing constraints, i.e. timing constraints that rule the minimum distance between commands issued to the same bank and to the same bank only (those who have the *RGB* attribute). For instance, if the channel scheduler executes an *activate* from a command register, then the corresponding bank scheduler must wait at least $dAW\text{-}RGB - 1$ cycles before inserting a *write* into the aforementioned command register. The -1 term reflects the fact that if a command is inserted into a command register in instant t_0, even in the best case scenario such command will only be executed by the channel scheduler in instant $t_0 + 1$.

4.3. Channel Scheduler

The channel scheduler has two functions: firstly, to arbitrate between and execute commands from the command registers, and secondly, to regularly refresh the SDRAM. The refreshing can be accomplished with a timer that triggers the execution of *refresh* commands at predetermined intervals [4]. Hence, this section will focus on the arbitration of commands from the command registers.

A block diagram of the channel scheduler is depicted in Fig. 4.2. Notice that the arbitration of commands is performed in two layers: firstly, commands are arbitrated inside their corresponding arbiters (in the figure, notice the demultiplexers used to route a command to the proper arbiter). More specifically, *write* and *read* commands are routed to the CAS Arbiter, while *activate* and *precharge* commands are routed to the AP Arbiter. Then, commands that won the arbitration in their corresponding first layer arbiters are arbitrated by the Command Bus Arbiter. The remainder of this section discusses each of these arbiters individually. Before the start of the discussion, however, the reader is again kindly reminded that bank schedulers handle only *exclusively intra-bank* timing constraints. All the remaining constraints are handled by the channel scheduler.

4.3.1. CAS Arbiter

The CAS Arbiter is where the *read/write* bundling is implemented. Its description in this section is structured into three parts: first, a discussion about the round-oriented operation of the CAS Arbiter is presented. Moreover, a classification of CAS commands with regard to the round-orientation is also introduced. Second, the transitioning be-

[4] For a thorough discussion within the scope of *refreshes* in *open-row* real-time SDRAM controllers, the interested reader can consult [127].

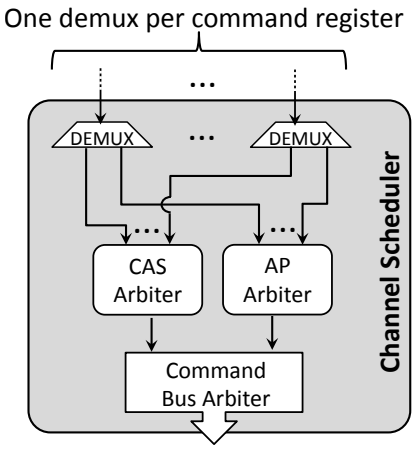


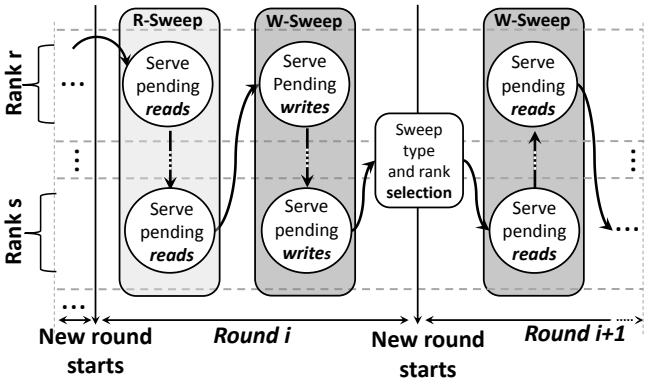
Figure 4.2.: Channel Scheduler.

tween consecutive rounds is addressed. As it will become clear, transitioning relies on the command classification mentioned in the previous part. Moreover, the transitioning includes the scheduling modification to handle *too-late* commands described in the beginning of this chapter. Finally, a logical architecture and an algorithmic description of the operation of the arbiter are provided.

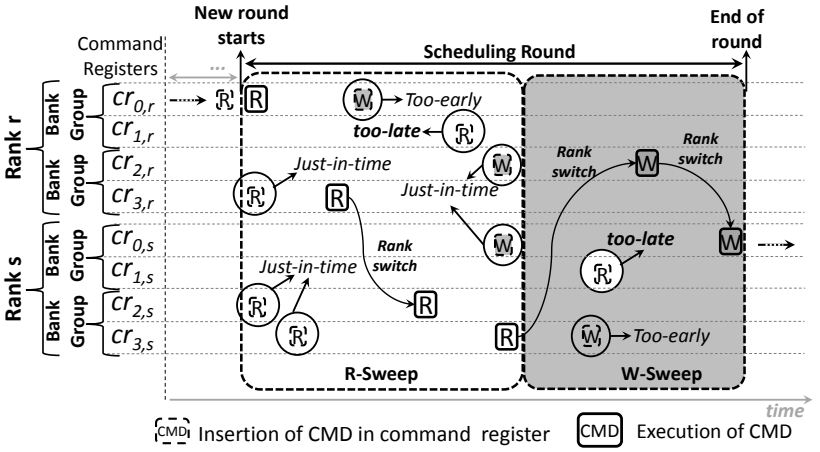
4.3.1.1. Round-Oriented Operation

The CAS arbiter operates with the concept of scheduling rounds. Its operation can be summarized with five rules:

1. in each scheduling round, at most one pending CAS command (regardless whether a *read* or a *write*) from each command register is selected and forwarded to the Command Bus Arbiter.
2. Each scheduling round is divided into a R-sweep and a W-sweep, as depicted in Fig. 4.3a. In a R-sweep, the arbiter serves at most one pending *read* for each command register of the rank currently being visited, a procedure referred to as visiting a rank. The R-sweep visits each rank at most one time and is over if all ranks have been visited. The W-sweep performs the same operation, but for *write* commands.
3. If a rank has no pertinent valid commands at the time it is visited (e.g. a rank with no pending *writes* is visited during a W-Sweep), it suffers a a so-called *empty* visitation, i.e. it is marked as visited.
4. If any two consecutive sweeps are of different types (i.e. W-Sweep followed by R-Sweep or vice-versa), then both sweeps visit ranks in the same order. For instance, in round i in Fig. 4.3a, the R-Sweep visits ranks using $r \rightarrow ... \rightarrow s$ order. Hence, the W-Sweep from round i also visits ranks using $r \rightarrow ... \rightarrow s$ order.
5. Within a rank visitation, the arbiter prioritizes the oldest command that is not blocked by rule (1) and that can be immediately executed.



(a) In terms of R- and W-Sweeps.



(b) In terms of commands and their classification.

Figure 4.3.: Example of read/write bundling. In (a), each circle represents a rank visitation. In (b), a hypothetical system with nR=2, nG=2 and nB=4 is assumed. Moreover, the data bus is omitted for the sake of clarity.

With regard to the round-oriented operation of the arbiter (which is guided by the aforementioned rules), the insertion of a CAS command into a command register can be classified into three types. Such types are exemplified in Fig. 4.3b and enumerated below:

- *Too-early*: the inserted command will not be considered for execution in the current scheduling round because a previous CAS command that occupied the same register was already executed in the current scheduling round, as dictated by rule (1). Commands that fall into this category are postponed until the next scheduling round.
- *Too-late*: the inserted command will not be executed in the current scheduling round because the insertion happened after the rank visitation that could execute such command has been performed. As the case for *too-early* commands, *too-late*

commands are also postponed until the next scheduling round. In Fig. 4.3b, the *read* commands in $cr_{1,r}$ and $cr_{1,s}$ fall into the *too-late* category. The former because it arrives after the rank switch from rank r to rank s in the R-Sweep. The latter because it arrives after the R-Sweep is over.

- *Just-in-time*: the inserted command will be executed in the current scheduling round.

Notice that the three types are mutually exclusive. This means, for instance, that a command can be either *too-late* or *too-early*, but not simultaneously *too-late* and *too-early*.

4.3.1.2. Round-to-Round Transitioning

When a scheduling round is over, the arbiter has to select how the next scheduling round will start. More specifically, the arbiter must decide if the next round starts with a R- or W-Sweep and which rank will be the first to be visited. In Fig. 4.3a, this dynamic decision is represented by the *sweep type and rank selection* block. The decision is implemented with Algorithm 1. In the algorithm, comments are written in gray after // symbols.

The algorithm receives as input a 4-tuple $(itype, irank, ilatereads, ilatewrites)$ and outputs a pair $(otype, orank)$. The input describes how the current scheduling round ended and whether *read* or *write* commands were inserted *too-late*. For instance, the input (`W`,r,`True`,`False`) describes a scheduling round that ended in a W-Sweep in rank r and that at least one *read* command (regardless of in which rank) was inserted *too-late*. The output describes how the new scheduling round should start and has the same semantics as the first two parameters of the input. For instance, $(\texttt{R}, r)$ means that the next scheduling round should start with a R-Sweep at rank r.

Notice that executing the algorithm with $(\texttt{W}, s, \texttt{True}, \texttt{False})$ as input tuple, which describes the scenario from Fig. 4.3b, results in $(otype, orank) = (\texttt{R}, r)$ as output. Notice also, however, that if the input is $(\texttt{W}, s, \texttt{False}, \texttt{False})$, i.e. a scenario similar to the one from Fig. 4.3b, but without the pending *too-late reads* from $cr_{1,r}$ and $cr_{1,s}$, the algorithm outputs $(otype, orank) = (\texttt{W}, s)$.

The algorithm constitutes the modification to handle *too-late* commands mentioned in the beginning of this chapter. As it will become clear in the timing analysis, given a CAS command that is *too-late* and that targets rank r where r could be any rank in the system, then such command will be blocked at most once by each competing command register in rank r (but twice for registers in other ranks). The analysis will also compute the influence that such algorithm has on *too-early* CAS commands. Finally, notice that *just-in-time* commands are always served in the scheduling round in which they are inserted and, as a consequence, experience smaller latencies than their *too-early* or *too-late* counterparts.

Algorithm 1 Determines how next scheduling round starts

```
1:
2: // Inputs
3: // itype: W or R (describes the type of the last sweep)
4: // irank: Index of the last visited rank
5: // ilatereads: True if there are pending too-late reads, False otherwise
6: // ilatewrites: True if there are pending too-late writes, False otherwise
7: //
8: // Outputs
9: // otype: W or R (describes whether the next round starts with a W- or R-Sweep.
10: // orank: Index of the next rank to be visited.
11: function SELECT_SWEEP_TYPE_AND_RANK(itype, irank, ilatereads, ilatewrites)
12:     if (itype = W and ilatereads = True) or (itype = R and ilatewrites = True)
    then
13:         otype ← TOGGLETYPE(itype)
14:         orank ← APPLYRULE4( ) // See Section 4.3.1.1
15:     else
16:         otype, orank ← itype, irank
17:     return otype, orank
18:
19: function TOGGLE_TYPE(type)
20:     if type = R then
21:         return W
22:     else
23:         return R
```

4.3.1.3. Logical Architecture and Algorithmic Description

In order to implement read/write bundling, the CAS Arbiter requires a state. As depicted in Fig. 4.4, the state is comprised of the *served flags* bit vector and the *current rank* and *bundling type* registers. The *current rank* register determines the rank currently being visited and the *bundling type* register determines whether the visitation is part of the R- or the W-sweep. Their values are updated according to the rules that guide sweep operations and rank visitations. The *served flags* are employed to enforce that at most one CAS command per command register is executed per round (first rule of operation). More specifically, the *served flags* are set to 0 at the beginning of the round. Then, once a command is executed, the flag from the corresponding command register is set to 1 and, as a consequence, any further command placed in such register is ignored by the CAS Arbiter until the next round.

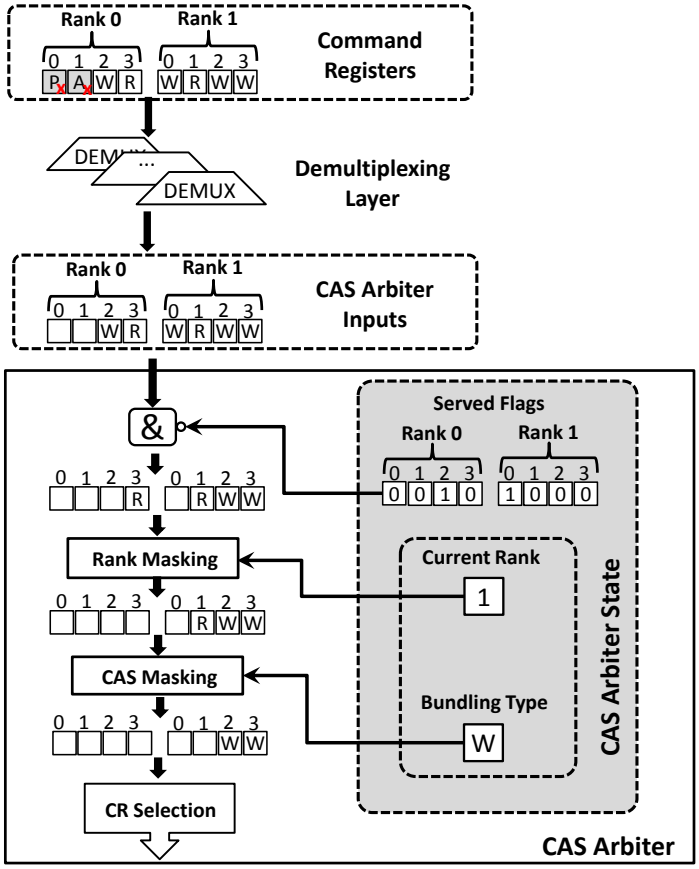


Figure 4.4.: Example of operation of the CAS Arbiter in a hypothetical system with $nR = 2$, $nB = 4$ and $nG = 1$. Notice that only CAS commands arrive at the input of the arbiter (*activates* and *precharges* are routed to the AP Arbiter). Moreover, notice that for the sake of simplicity, the *bank groups* feature and the logic that updates the state of the arbiter are omitted.

Selecting the command that wins the arbitration consists firstly in masking out commands that do not conform to the state. More specifically, commands in registers whose *served flag* is set to 1 and whose type and rank do not match the *current rank* and *bundling type* registers. Then, the commands that are not masked out go through the CR Selection block, which simply prioritizes the oldest CAS command. Notice that in systems with $nG > 1$, prioritizing the oldest command might lead to the execution of consecutive CAS commands that target the same *bank group*, a behavior that will be accounted for in the timing analysis.

In order to avoid misunderstandings, the operation of the CAS Arbiter is formalized with Algorithm 2 using pseudo-code. The following observations about the pseudo-code are made: firstly, comments are depicted with grey font after // symbols. Moreover, the pseudo-code is heavily commented for ease of understanding. Secondly, the pseudo-code uses a objected-oriented notation. For instance, *cmd.rank* and *cmd.bank* are used respectively to refer to the target bank and the target rank of a command. Thirdly, code that initializes the control registers has been left out. Fourthly, the difference between a procedure and a function is that the latter returns a value (to be employed by the function caller), while the former returns nothing. And finally, the OPERATE( ) procedure describes the operation of the CAS Arbiter (which consists of performing scheduling rounds).

Algorithm 2 CAS Arbiter Operation

```
1:  CAS Arbiter Control Registers
2:      // parameters of SDRAM module configuration
3:      nR, nG, nB;
4:      // For sweeping and rank visitation control
5:      served_flags[nR][nB], current_rank, bundling_type;
6:      last_exec_cmd; // Last executed CAS command (regardless of the rank)
7:  end CAS Arbiter Control Registers
8:
9:  List of Command Registers
10:     crs[nR][nB];
11: end List of Command Registers
12:
13: // Basic operation of CAS Arbiter
14: procedure OPERATE( )
15:     while True do
16:         PREPARE_FOR_NEXT_ROUND( )
17:         PERFORM_SCHEDULING_ROUND( )
18:
19: // Continues in next page...
```

```
20: procedure PREPARE_FOR_NEXT_ROUND( )
21:     // Determines how the next round starts
22:     itype ← last_exec_cmd.type
23:     irank ← last_exec_cmd.rank
24:     ilatereads ← ARE_THERE_LATE_READS( )
25:     ilatewrites ← ARE_THERE_LATE_WRITES( )
26:     bundling_type, current_rank ← SELECT_SWEEP_TYPE_AND_RANK(itype,
  irank, ilatereads, ilatewrites) // See Algorithm 1
27:
28:     // Resets (clears) the served_flags vector
29:     for i ← 0; i < nR; i ← i + 1 do;
30:         for j ← 0; j < nB; j ← j + 1 do;
31:             server_flags[i][j] ← 0;
32:
33: procedure PERFORM_SCHEDULING_ROUND( )
34:     // Keeps track of the first visited rank
35:     first_visited_rank ← current_rank
36:
37:     // First Sweep operation of the round
38:     // Example: in a system with nR=4, if the first rank to be visited is 2, the arbiter
39:     // visits ranks using the following order: 2 → 3 → 0 → 1
40:     do
41:         VISIT_RANK( ) // Visits the current rank
42:         current_rank ← (current_rank + 1) mod nR
43:     while current_rank ≠ first_visited_rank
44:
45:     // After the first sweep, bundling_type register is toggled...
46:     bundling_type ← TOGGLETYPE(bundling_type) // See Algorithm 1
47:
48:     // Second Sweep operation of the round (at this point, current_rank is equal to
49:     // first_visited_rank). Notice that if the first sweep visited ranks
50:     // using 2 → 3 → 0 → 1 order, so should the second sweep.
51:     do
52:         VISIT_RANK( ) // Visits the current rank
53:         current_rank ← (current_rank + 1) mod nR
54:     while current_rank ≠ first_visited_rank
55:     return
56:
57: function ARE_THERE_TOO_LATE_READS( )
58:     retval = False
59:     for i ← 0; i < nR; i ← i + 1 do;
60:         for j ← 0; j < nB; j ← j + 1 do;
61:             // Continues in next page...
```

```
62:             // A read is too-late if the round is over and the served flag
63:             // of the corresponding register is unset.
64:             if crs[i][j].cmd = R and served_flags[i][j] = 0 then
65:                 retval ← True
66:             else
67:                 retval ← False
68:     return retval
69:
70: function Are_There_Too_Late_Writes( )
71:     retval = False
72:     for i ← 0; i < nR; i ← i + 1 do;
73:         for j ← 0; j < nB; j ← j + 1 do;
74:             // A write is too-late if the round is over and the served flag
75:             // of the corresponding register is unset.
76:             if crs[i][j].cmd = W and served_flags[i][j] = 0 then
77:                 retval ← True
78:             else
79:                 retval ← False
80:     return retval
81:
82: // The rank to be visited and the sweep type are determined by the current_rank
83: // and bundling_type registers. Notice that empty visitations happen when None
84: // is returned the first time Find_Next_CAS( ) is called.
85: procedure Visit_Rank( )
86:     winningcmd ← Find_Next_CAS( )
87:
88:     while winningcmd ≠ None do
89:         while (winningcmd cannot be executed without violating a
    constraint) do
90:             wait until next cycle
91:         Execute_Command(winningcmd)
92:         winningcmd ← Find_Next_CAS( )
93:
94:     // If this point is reached, then the visitation is over
95:     return
96:
97: // Finds the next CAS that matches bundling_type and current_rank
98: // (Returns None if a rank visitation is over)
99: function Find_Next_CAS( )
100:     oldest_cmd ← None
101:     for i ← 0; i < nB; i ← i + 1 do;
102:         cmd ←crs[current_rank][i].cmd // Gets command from cmd. register
103:         // Continues in next page...
```

```
104:         if cmd.type = bundling_type and served_flags[current_rank][i] = 0 then
105:             oldest_cmd ← SELECT_OLDEST(oldest_cmd, cmd)
106:     return oldest_cmd
107:
108: procedure EXECUTE_COMMAND(cmd)
109:     SEND_COMMAND_TO_COMMAND_BUS_ARBITER(cmd)
110:
111:     // As CAS commands have priority over activates and precharges they
112:     // are immediately executed and, hence, control registers
113:     // can be updated.
114:     served_flags[cmd.rank][cmd.bank] ← 1
115:     last_exec_cmd ← cmd
116:
117:     return
118:
119: // If cmda = None and cmdb = None, this function also returns None.
120: function SELECT_OLDEST(cmda,cmdb)
121:     if cmda = None then
122:         retcmd ← cmdb
123:     else if cmdb = None then
124:         retcmd ← cmda
125:     else
126:         // Compares the insertion timestamps (in the corresponding registers)
127:         if cmda.insertiontimestamp < cmdb.insertiontimestamp then
128:             retcmd ← cmda
129:         else
130:             retcmd ← cmdb
131:     return retcmd
```

4.3.2. AP Arbiter

The operation of the AP Arbiter can be logically divided into two arbiters: module-level and rank-level.

The module-level arbitration is firstly discussed. In scenarios in which $nR = 1$, the *activate* or *precharge* command that wins the rank-level arbitration is consequently also the winner of the module-level arbitration. In scenarios in which nR is larger than 1, the module-level arbiter must select a command among the winners of the corresponding rank-level arbiters. For that purpose, it employs round-robin.

The rank-level arbitration of the AP Arbiter is now discussed. Inside a rank, the AP Arbiter must take the following into account: 1) the execution of *activate* commands is constrained by $dAA\text{-}R\overline{B}$ (with the G or $\overline{G}$ arguments for DDR4) and by the t_{FAW} constraint (see Section 2.3). And 2), *precharges* can be executed back-to-back, i.e. one per cycle.

In systems in which $nG = 1$, the rank-level arbitration of the AP Arbiter follows a simple scheduling rule: the oldest *activate* or *precharge* that can be immediately executed without violating a constraint is given priority.

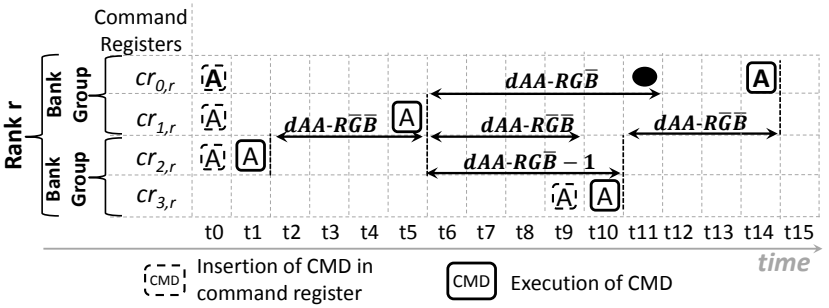
In systems in which $nG > 1$, the rank-level arbitration of the AP Arbiter is slightly more complex: firstly, an *activate* command is selected using what this dissertation calls *real-time aware oldest ready* arbitration. If such *activate* is older than the oldest pending *precharge* (or if there are no pending *precharges*), than the *activate* is the winner of the rank-level arbitration. Otherwise, the oldest pending *precharge* (if it exists) wins the rank-level arbitration.

The *real-time aware oldest ready* arbitration is now discussed. The *oldest ready* expression means the oldest *activate* that can be immediately executed has priority. The *real-time aware* expression means that the *ready* requirement (being able to be immediately executed) is ignored in one specific situation: if more than $dAA\text{-}R\overline{GB}$ cycles have passed since the last execution of an *activate* command. Such exception is to prevent the situation depicted in Fig. 4.5a.

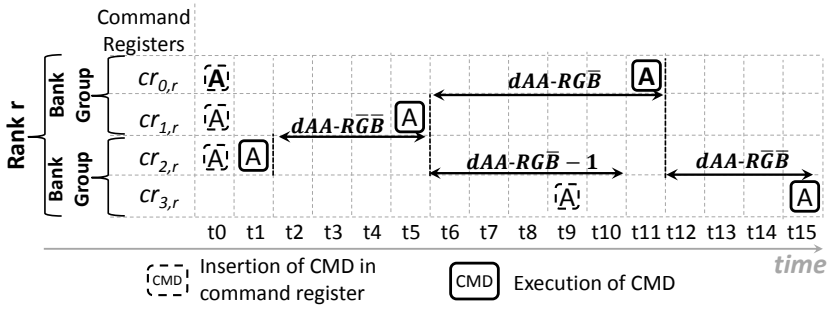
In the figure, the *activate* in $cr_{0,r}$ is blocked by three interfering *activates*. However, because the *activate* in $cr_{3,r}$ arrives only in $t9$, the time interval between its execution with regard to the *activate* from $cr_{1,r}$ is larger than $dAA\text{-}R\overline{GB}$. From the worst-case latency perspective, this would mean that, inside its own rank, an *activate* command could be blocked by $nB - 1$ interfering *activates* and that each of this blockings would amount to $dAA\text{-}RG\overline{B} - 1$.

Hence, the AP Arbiter employs the *real-time aware oldest ready* arbitration, which is depicted in Fig. 4.5b. In the figure, notice that the *activate* in $cr_{0,r}$ is only blocked by $nB - 2$ interfering *activates*. More importantly, the only way for it to be blocked $nB-1$ times would be if all interfering *activates* were previously available, in which case a blocking of $dAA\text{-}R\overline{GB} \cdot (nB - 1)$ cycles would be observed.

Finally, in order to avoid misunderstandings, the operation of the AP Arbiter is formalized using pseudo-code in Algorithm 3. From the perspective of the pseudo-code



(a) Oldest Ready arbitration.



(b) Real-time aware oldest ready arbitration.

Figure 4.5.: Intra-rank arbitration of *activates* in a scenario in which $dAA\text{-}R\overline{GB} = 6$ and $dAA\text{-}R\overline{GB} = 4$. The effect of t_{FAW} is deliberately ignored, but is accounted for in the timing analysis. Moreover, in (a), the solid black circle represents the moment in time in which the *activate* from $cr_{0,r}$ would be executed if the *ready* requirement was ignored.

employed in Algorithm 3, the same observations as for Algorithm 2 (see Section 4.3.1.3) apply.

Algorithm 3 Operation of the AP Arbiter

```
1:  AP Arbiter Control Registers
2:      last_act[nR]; // Holds the last executed activate (per rank)
3:      last_exec_cmd; // Last executed activate or precharge (regardless of the rank)
4:      current_clock_tick; // Keeps track of time
5:  end AP Arbiter Control Registers
6:
7:  List of Command Registers
8:      crs[nR][nB];
9:  end List of Command Registers
10:
11: // Describes basic operation of the AP Arbiter
12: procedure OPERATE( )
13:     while True do
14:         cmd ← FIND_NEXT_CMD_TO_BE_EXECUTED( )
15:         if cmd ≠ None then
16:             SEND_COMMAND_TO_COMMAND_BUS_ARBITER(cmd)
17:             // In the Command Bus Arbiter Arbiter, cmd can be blocked by other
18:             // CAS commands. Hence, the AP arbiter waits....
19:             wait until command is executed
20:             // Now the control registers are updated...
21:             if cmd.type = Activate then
22:                 last_act[cmd.rank] ← cmd
23:             last_exec_cmd ← cmd
24:
25: // Module-level arbitration: searches all ranks and returns an
26: // activate or precharge that can be immediately executed without violating
27: // a constraint (or None, if no commands are found).
28: function FIND_NEXT_CMD_TO_BE_EXECUTED( )
29:     // Searches ranks using round-robin.
30:     winningcmd ← None
31:     aux ← (last_exec_cmd.rank + 1) mod nR
32:     // Continues in next page...
```

```
33:     do
34:         winningcmd ← FIND_NEXT_CMD_FROM_RANK(aux)
35:         if winningcmd ≠ None then
36:             break
37:         aux ← (aux + 1) mod nR
38:     while aux ≠ last_exec_cmd.rank
39:
40:     return winningcmd
41:
42: // Rank-level arbitration.
43: function FIND_NEXT_CMD_FROM_RANK(rankindex)
44:     activate_command ←FIND_NEXT_ACT_IN_RANK(rankindex)
45:     precharge_command ←FIND_NEXT_PRE_IN_RANK(rankindex)
46:
47:     if activate_command can be immediately executed without violating a
      constraint then
48:         winningcmd ← SELECT_OLDEST(activate_command, precharge_command)
49:     else
50:         winningcmd ← precharge_command
51:     return winningcmd
52:
53: // Selects the oldest precharge from an specific rank.
54: // Returns None if no precharges are found.
55: function FIND_NEXT_PRE_IN_RANK(rank_id)
56:     winningcmd ← None
57:     for b ← 0; b < nB; b ← b + 1 do;
58:         winningcmd ← SELECT_OLDEST(winningcmd, crs[rank_id][b])
59:     return winningcmd
60:
61: // Selects an activate command from an specific rank using
62: // real-time aware oldest ready arbitration.
63: // Returns None if no activates are found.
64: function FIND_NEXT_ACT_IN_RANK(rank_id)
65:     // This variable will hold the oldest pending activate in the
66:     // same bank group (sbg) as the last activate executed in rank_id
67:     oldest_cmd_sbg ← None
68:     // This variable will hold the oldest pending activate
69:     // in a different bank group (dbg) as the last activate executed in rank_id
70:     oldest_cmd_dbg ← None
71:
72:     // Continues in next page...
```

```
73:     // Now the oldest_cmd_sbg and oldest_cmd_dbg variables are filled
74:     // For that purpose, the arbiter iterates over the command registers
75:     // from the rank under consideration
76:     for i ← 0; i < nB; i ← i + 1 do;
77:         cmd ←crs[rank_id][i].cmd // Gets the command from the command register
78:         if cmd ≠ None then
79:             if cmd.type = Activate then
80:                 if bank i in same group as bank from last_cmd[rank_id] then
81:                     oldest_cmd_sbg ← SELECT_OLDEST(oldest_cmd_sbg, cmd)
82:                 else
83:                     oldest_cmd_dbg ← SELECT_OLDEST(oldest_cmd_dbg, cmd)
84:
85:     if oldest_cmd_dbg = None then
86:         winningcmd ← oldest_cmd_sbg
87:     else if oldest_cmd_sbg = None then
88:         winningcmd ← oldest_cmd_dbg
89:     else
90:         // If this point is reached, both oldest_cmd_sbg and oldest_cmd_dbg
91:         // are different than None.
92:         // The real-time aware oldest ready arbitration is performed here.
93:         if (oldest_cmd_sbg is older than oldest_cmd_dbg and
94:            (current_clock_tick − last_exec_cmd.execution_timestamp) > dAA-RḠB̄) then
95:             winningcmd ← oldest_cmd_sbg
96:         else
97:             winningcmd ← oldest_cmd_dbg
98:     return winningcmd
99:
100: // If cmda = None and cmdb = None, this function also returns None.
101: function SELECT_OLDEST(cmda,cmdb)
102:     if cmda = None then
103:         retcmd ← cmdb
104:     else if cmdb = None then
105:         retcmd ← cmda
106:     else
107:         // Compares the insertion timestamps (in the corresponding command registers)
108:         if cmda.insertiontimestamp < cmdb.insertiontimestamp then
109:             retcmd ← cmda
110:         else
111:             retcmd ← cmdb
112:     return retcmd
```

4.3.3. Command Bus Arbiter

The command bus can only carry one command per cycle and, hence, needs to be arbitrated. As the CAS Arbiter and the AP Arbiter only forward commands that can be immediately executed, the Command Bus Arbiter employs a simple fixed priority scheme: commands from the CAS Arbiter have priority over *activates* and *precharges*. Hence, *reads* and *writes* do not suffer any interference.

4.3.4. Hardware Implementation

From the perspective of hardware design, bank schedulers can be implemented with state machines and request queues use a combination of state machines and sequential elements. Hence, the challenging portions of the controller are the channel scheduler[5] and the PHY layer. As the PHY layer is orthogonal to command scheduling, it can be developed without any regard for mixed criticality and, hence, its implementation has not been investigated in this dissertation. The channel scheduler, however, is crucial for the suitability of the controller to mixed criticality environments.

In order to understand why its hardware implementation is challenging, the reader is kindly reminded that the channel scheduler must operate at the same frequency employed by the data bus of the controlled SDRAM module. This is because the timing constraints are measured in terms of data bus clock cycles.

Due to the complexity of the channel scheduler, fitting all the logic necessary to make a command scheduling decision into a single clock cycle is not an effective solution, as the critical path would compromise the maximum operating frequency of the circuit. A better approach is to employ a pipelined architecture, i.e. partition the decision into two or more clock cycles. From the *best-effort* perspective, the pipelining also has negligible impact on data bus utilisation assuming a system backlogged with commands. From the real-time perspective, the latency incurred by the pipeline depth is negligible in comparison with inter-bank and inter-rank interference.

Moreover, such latency can be mostly hidden by inserting commands earlier in a command register. For instance, consider that a bank scheduler is serving a read request that missed at the *row buffer*, which demands a P-A-R sequence. After the *precharge* command is executed, the bank scheduler has to wait $dPA\text{-}RGB - 1$ cycles before inserting the *activate* into the corresponding command register (see Section 4.2). If the pipeline depth of the AP Arbiter is 4 cycles, the bank scheduler can perform the insertion of the *activate* as soon as $dPA\text{-}RGB - 1 - 4$ cycles after the execution of the *precharge*.

The initial results of the pipelined implementation have been published in [23]. However, such article had the following limitations:

- It did not consider the modification to handle *too-late* CAS commands mentioned in the introduction of this chapter.

[5] Because the channel scheduler uses the *age* of a command (i.e. how old a command is with regard to other commands) to make scheduling decisions, the channel scheduler hardware implementation includes the command registers.

- It did not consider SDRAM modules built using devices in which nG is larger than one (i.e. DDR4).

Those shortcomings were addressed in a master thesis [123]. The interested reader is consequently referred to [123] for further micro-architecture details. However, synthesis results obtained with a 28 nm TSMC process and the Synopsys Design Compiler tool are presented in Fig. 4.6 and in Fig. 4.7. The former refers to frequency and the latter to area. The results refer to a pipeline depth of 4 stages.

The results can be summarized with the following observations:

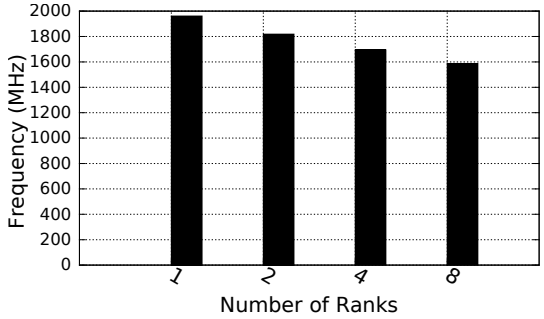
- The larger the number of banks in the controlled SDRAM module (more specifically, the larger $nR \cdot nB$), the lower the operating frequency and the larger the area of the channel scheduler.
- The *bank groups* feature had negligible overhead in the frequency of the synthesized design. It did, however, lead to a small increase in area.
- Last and most importantly, the obtained frequency results cover all speed bins of DDR2, DDR3 and DDR4 SDRAM devices. This is the case even for scenarios with 4 or 8 ranks, which are uncommon in the embedded world due to their cost and power consumption.

4.4. Requestor-to-Bank Assignment

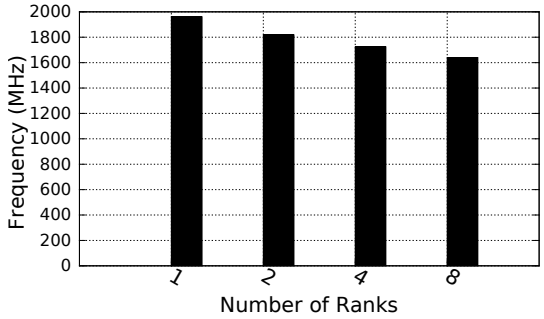
The main focus of this dissertation is on the design of a channel scheduler that enforces sufficient independence between all banks in the controlled SDRAM module. Simply stated, the channel scheduler is carefully designed to enforce that the inter-bank interference that any command observes is always predictable. However, the channel scheduler alone is not enough in order to enforce sufficient temporal and spatial independence between applications from different criticalities, as dictated by safety standards, e.g. [52]. The reason for it is intra-bank interference. More specifically, if different requestors compete for the same bank, they can exert interference on each other. (Notice that the word requestor refers to a core executing an application).

Intra-bank interference can be controlled by implementing a bank partitioning scheme that defines the physical SDRAM banks to which a core has access. The implementation of partitioning schemes has been the subject of many articles, e.g. [128, 85]. In summary, the partitioning is generally built on top of a virtual memory layer, which requires Operating System (OS) support and dedicated hardware, i.e. a Memory Management Unit (MMU).

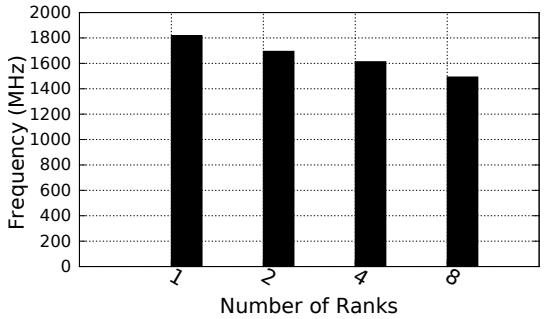
This section, however, focuses on the implication that the partitioning has on the timing and safety of *critical* applications. For that purpose, consider Fig. 4.8, which depicts possible ways to assign cores to SDRAM banks. The figure assumes that two cores are dedicated to the execution of *critical* applications, while two other cores execute *best-effort* applications. Moreover, the ellipses in the bottom of the figure classify the banks into three categories, according to their corresponding core-assignment. The categories are enumerated below:



(a) $nB = 8$, $nG = 1$ (DDR2 and DDR3).

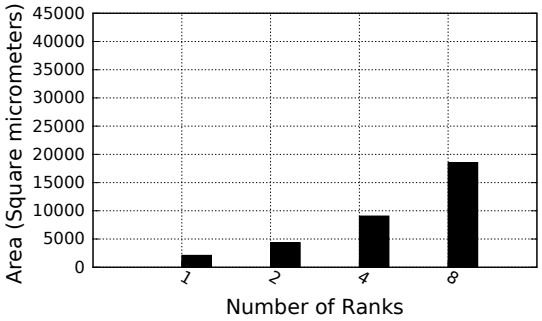


(b) $nB = 8$, $nG = 2$ (DDR4).

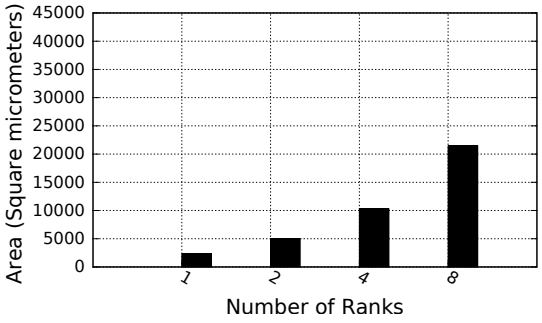


(c) $nB = 16$, $nG = 4$ (DDR4).

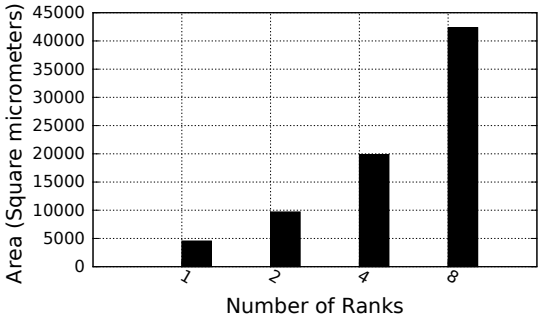
Figure 4.6.: Frequency results obtained synthesizing the channel scheduler for a 28 nm TSMC process. Notice that all figures employ the same scale. Notice also that in each of the subfigures, the nB and nG parameters are fixed. For each pair of nB and nG, results using different values of nR are depicted.



(a) $nB = 8$, $nG = 1$ (DDR2 and DDR3).



(b) $nB = 8$, $nG = 2$ (DDR4).



(c) $nB = 16$, $nG = 4$ (DDR4).

Figure 4.7.: Area results obtained synthesizing the channel scheduler for a 28 nm TSMC process. The same observations from Fig. 4.6 apply.

1. *Private bank*: refers to banks that are used by a single core, e.g. banks 0 and 1 in Fig. 4.8.
2. *Shared bank (within same criticality)*: refers to banks that are used by two or more cores that run applications from the same criticality level, e.g. banks 3 and 5 in Fig. 4.8.
3. *Shared bank (across different criticalities)*: refers to banks that are used by two or more cores that run applications from different criticality levels, e.g. bank 4 in Fig. 4.8.

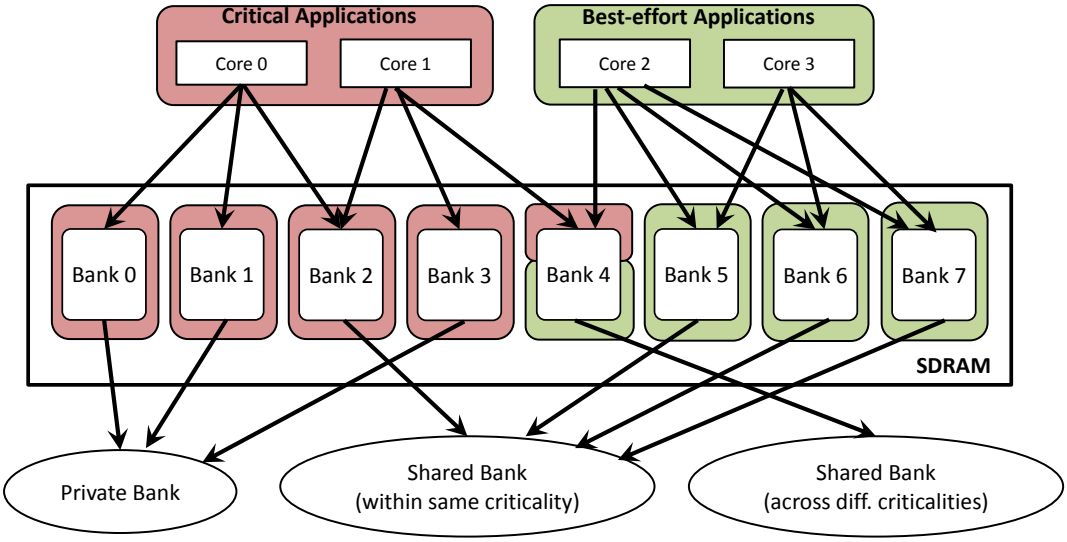


Figure 4.8.: Requestor-to-bank assignment and classification. The background color of a bank represents the criticality of the applications using such bank. For instance, bank 0 has a red background because it is only used by *critical* applications. Bank 4, however, has a red and green background because it is used by *critical* and *best-effort* applications.

Notice that a core can have access to banks from different categories. For instance, in Fig. 4.8, core 1 accesses bank 3 (which is private), shares bank 2 with other *critical* core and shares bank 4 with *best-effort* cores. The remaining of this section focuses on describing the impact that these requestor-to-bank assignments have on spatial and temporal isolation for a *critical* application. For that purpose, the reader is again reminded that safety standards such as [52] call for *sufficient independence* between applications from different criticalities (see Section 1.1 in the Chapter 1).

Private banks are firstly discussed. Accessing a *private* bank is fully in consonance with what safety standards require. The *sufficient independence* is established because in addition to not observing intra-bank interference, an application accessing a *private* bank only observes a controlled and upper-bounded amount of inter-bank interference (because of the channel scheduler). Moreover, as it will become clear, *private* banks are specially interesting for *critical* applications, because the effect of *row buffer* locality

can be taken into account when computing timing guarantees for such application (see *private*-bank assumption in Section 3.2 from Chapter 3).

Banks shared within the same criticality are now addressed. Accessing a shared bank with other *critical* core is also in consonance with requirements of safety standards. This is because the standards dictate *sufficient independence* between requestors of different criticality levels (and not between requestors of the same criticality). As *critical* applications go through a certification process, their behavior is well understood. As a consequence, one can assume they will not corrupt data in the shared bank. Moreover, the temporal interference they can exert on each other can be accounted for in a timing analysis. Notice, however, that two *critical* applications running simultaneously in different cores and sharing a bank will certainly disturb the *row buffer* locality of such bank, as it will be addressed in Chapter 5.

Finally, banks shared across different criticalities are discussed. With respect to it, notice that this sharing category definitely has its place in mixed criticality systems because, as pointed out in [29], subfunctions of *critical* applications might also be required by non-*critical* applications. For instance, data from wheel sensors in a car are used by not only the Anti-lock Braking System (ABS) (*critical*), but also by the navigation system (non-*critical*).

However, unlike the two previous categories, bank sharing across criticalities demands further measures in order to comply with requirements of safety standards. This is because *best-effort* applications do not go through a certification process. Consequently, their behavior is unpredictable and they might contain bugs, which break the so-called *sufficient* spatial and temporal isolation of *critical* applications. As an example, consider a *best-effort* application running on a superscalar processor that tolerates multiple pending memory requests. A bug in such application can lead to an endless burst of requests to a shared bank, which would as a consequence flood the corresponding bank request queue of the controller (see Fig. 4.1 on page 48), thus unpredictably disturbing the timing of *critical* applications.

The author of this dissertation envisions two solutions to allow safe bank sharing across a criticality level. The first one is to employ an online monitoring approach [131, 130, 95, 94] that constantly checks whether each *best-effort* application sticks to the expected behavior. In [131, 130], for instance, the expected behavior refers to a maximum number of memory requests issued in a predetermined time frame.

If the monitor, which can be implemented as part of the OS, detects a deviation, it can take action (e.g. either throttle the number of requests made by the *best-effort* application or shut it off). In practical terms, the monitor enforces that the *best-effort* application behaves in a predictable fashion, thus allowing such predictable behavior to be taken into account to compute timing guarantees for the *critical* applications, reestablishing sufficient temporal isolation. In the remaining of this dissertation, the monitoring solution is considered instead of the second solution, which is addressed in the next paragraph.

The second solution is to modify the architecture of the controller, adding two bank

request queues per bank, one for *critical* requests and another for *best-effort* requests. Each bank scheduler then prioritizes requests from the *critical* request queue. For the sake of clarity, an example is depicted in Fig. 4.9. Notice that this would demand that each incoming request contains enough information to identify whether the core issuing the request is running *critical* or *best-effort* applications.

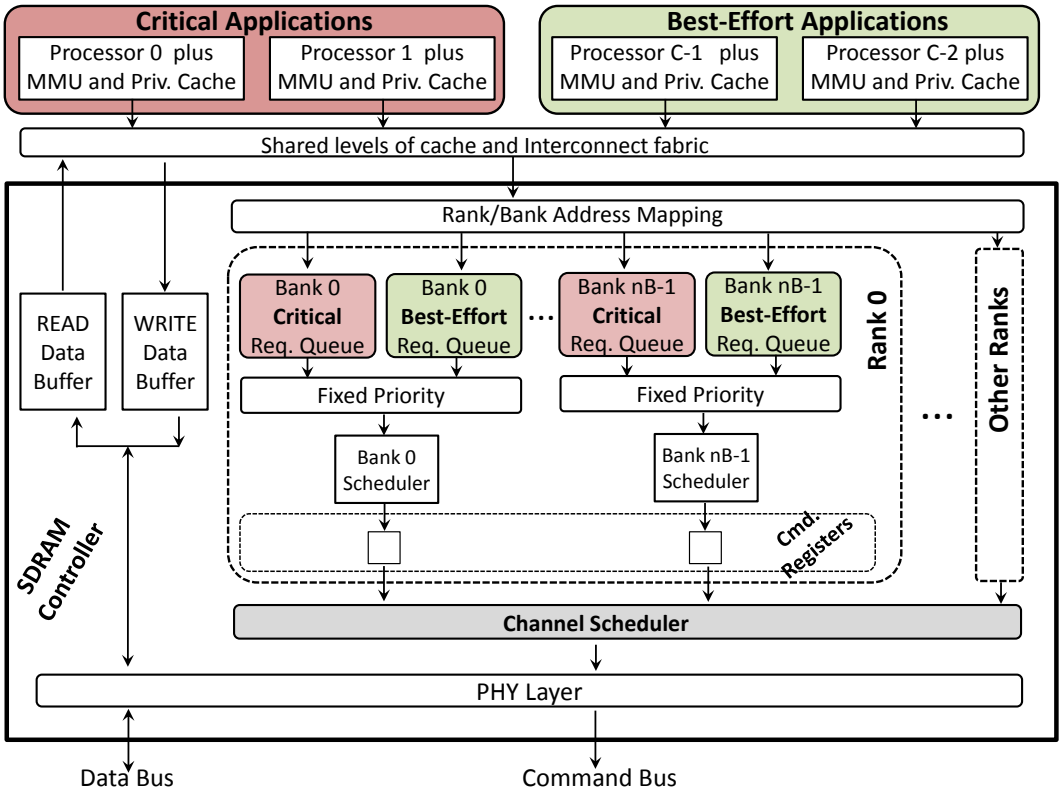


Figure 4.9.: Modified controller architecture to allow safe bank sharing between *critical* and *best-effort* requestors.

Notice also that for both solutions, data integrity can be enforced at the level of virtual memory. More specifically, virtual memory pages of *best-effort* applications that are translated into physical addresses containing *critical* data should be marked as non-writable. In simpler systems with no virtual memory, it is also possible to enforce data integrity with a Memory Protection Unit (MPU).

4.5. Summary

This chapter proposes a non-pattern-based multi-generation SDRAM controller architecture for mixed criticality environments. The controller supports two level of criticalities: *critical* and *best-effort*. The expectations from both levels with regard to timing are well described in Section 1.1.1.

Inside the controller, the distinction between a *critical* and a *best-effort* requestors

happen at the intra-bank level. In *best-effort* banks, aggressive request reordering is employed in order to increase the exploitation of *row buffer* locality, i.e. the FR-FCFS policy is employed. In *critical* banks (which include banks shared by *critical* and *best-effort* requestors), no reordering is implemented. Once a request is translated into commands, however, such commands compete equally with all other pending commands in the system (regardless of the criticality).

Pending commands are arbitrated in two layers: firstly, within their own type arbiters (CAS and AP Arbiters), and then at the Command Bus Arbiter. The most important part of the scheduling is done in the CAS Arbiter, which uses read/write bundling in order to minimize data bus turnarounds and rank switching events. As discussed in Chapter 2, both can severely decrease data bus utilisation and, as a consequence, hurt real-time guarantees and decrease average performance.

The scheduling performed by the controller is described in terms of minimum distances between consecutive commands using the notation discussed in Section 2.3. The channel scheduler, the most crucial part of the controller from the perspective of mixed criticality, has been synthesized at competitive operational frequencies using a 28 nm TSMC process.

Finally, considerations about the mapping of requestors to SDRAM banks are also presented. More specifically, the channel scheduler is designed to bound inter-bank interference. However, providing sufficient independence for *critical* requestors also requires bounding intra-bank interference. The latter is achieved with a proper requestor-to-SDRAM-bank assignment. In addition to the proper assignment, safely sharing a bank between *critical* and *best-effort* requestors also requires either monitoring the behavior of the latter during run-time or a small architectural modification in the controller (having two request buffers per bank).

5. Timing Analysis

This chapter presents an analysis of the temporal properties of the scheduling performed by the controller proposed in Chapter 4.

More specifically, the analysis presented in this chapter computes the worst-case cumulative SDRAM latency of a *critical* task (L_{Task}^{SDRAM}), i.e. the the maximum amount of time that a *critical* task spends idle while waiting for its memory requests to be served. The analysis is generic and works regardless of the DDR generation and configuration of the SDRAM module (nR, nG and nB). For the sake of notation, SDRAM generations that do not have the *bank groups* feature, e.g. DDR2 and DDR3, are considered to have $nG = 1$.

The remaining of this chapter is structured as follows: Section 5.1 presents the assumptions made by the analysis. Section 5.2 computes the worst-case latency of individual SDRAM commands. Section 5.3 uses the computed command latencies to calculate the worst-case latency of requests. Section 5.4 combines the request latencies to compute the worst-case SDRAM latency of a task. Section 5.5 makes considerations about bank sharing. Finally, Section 5.6 summarizes the chapter.

5.1. Assumptions

The timing analysis relies on the following assumptions:

1. the processor running the task *under analysis* (u.a.) relies on caches and only accesses the SDRAM to retrieve or forward cache lines.
2. The processor is fully timing compositional [125], which means that it uses in-order execution and stalls at every read request.
3. The *write-buffer* between the cache and SDRAM is disabled and, hence, the processor also stalls at write requests[1]. In the ARMv8-A architecture [101], for instance, this is achieved by disabling the *early write acknowledgment* feature. This enforces that a request only arrives at the SDRAM controller after the previous request (from the same task) has been served.
4. No multi-threading/context switches occur due to task scheduling. This enforces that no cache related effects change the number of cache misses experienced by the task u.a..
5. The processor running the task u.a. has exclusive access to one of the banks

[1] Notice that the presence of a *write-buffer* allows a processor to keep executing while a write request is being processed, potentially hiding the latency of a write request. Consequently, making a no *write-buffer* assumption is conservative.

(*private*-bank assumption discussed in Section 3.2) and the corresponding bank request queue and bank scheduler employ the FCFS policy and the *open-row* policy, respectively.

Three important remarks are made about the assumptions: firstly, notice that none of them include having knowledge about interfering tasks on the system. This is a desirable feature, because the computed bound remains valid regardless of activity of interfering requestors. Secondly, computing a cumulative bound (i.e. over all requests performed by a task) using a *private*-bank assumption allows the effect of *row buffer* locality to be captured.

And finally, notice that if requestors (i.e. processors running applications) need to share data, or if the number of SDRAM banks in the system is smaller than the number of requestors, one or more SDRAM banks will have to be shared (see Section 4.4), which violates assumption 5. Computing timing bounds for requests that target shared SDRAM banks is discussed separately in Section 5.5.

5.2. Worst-case Latency of Commands

The worst-case latency of a command refers to the largest observable timing interval between the insertion of a command into a command register and its execution by the channel scheduler. This section calculates the worst-case latencies of *read*, *activate* and *precharge* commands. The case for *write* commands is symmetrical to the case for *read* commands and, hence, equations for it are only available in Appendix B.

In order to avoid confusion, the following remarks about the conventions employed in this section are made:

- The command register that contains the command under analysis is referred to as *cr u.a.*.
- Other command registers in the system are referred to as $icr_{b,r}$, where b is a bank index and r is a rank index.
- The rank that contains the *cr u.a.* is referred to as the *rank u.a.*.
- $\max\begin{cases} A \\ B \end{cases}$ returns the largest value between A and B, which is also represented with $\max\{A, B\}$.
- $\begin{cases} \textbf{if } cond \textbf{ then: } A \\ \textbf{else then: } B \end{cases}$ returns A if *cond* is true, and B otherwise.

Moreover, in order to improve reading guidance, the notations used to refer to command latencies are summarized in Table 5.1. Notice that for CAS commands, two types of worst-case latency are considered: 1) the one experienced if the CAS u.a. succeeds a CAS command (SC) in the *cr u.a.* (i.e. is inserted into *cr u.a.* after a CAS command). 2) The one experienced if the CAS u.a. succeeds a non-CAS (SNC) command in *cr u.a.*. This distinction enables a tighter analysis of the round-oriented operation of the CAS arbiter.

The remaining of this section firstly computes the worst-case latency of *read* commands and then computes the worst-case latency of *activate* and *precharge* commands.

Table 5.1.: Notation used for worst-case latencies of SDRAM commands.

.

Worst-case Latency Notation	SDRAM Command u.a.	Succeeding a (following a)	Computed according to
L^R_{SC}	R	R or W	Theorem 5.8
L^R_{SNC}	R	A	
L^W_{SC}	W	R	Theorem B.5 in Appendix B
L^W_{SNC}	W	A	
L^A	A	P	Theorem 5.10
L^P	P	A or W or R	Theorem 5.11

5.2.1. Worst-case Latency of Read Commands

A conservative bound on the worst-case latency of a *read* command needs to consider the *too-late* and the *too-early* cases. (The *just-in-time* case is dominated by the *too-late* case, as *just-in-time* CAS commands are executed in the same round in which they are inserted into the *cr u.a.*).

That being stated, the computation of the worst-case latency of a *read* command is structured as follows: firstly, in Section 5.2.1.1, an assisting lemma that captures the effect of *group interleaving* in CAS command scheduling is stated and proven. Moreover, other initial considerations are also presented. Secondly, in Section 5.2.1.2, the worst-case latency of a *too-early read* command is computed. Then, in Section 5.2.1.3, the worst-case latency of a *too-late read* command is computed. Finally, in Section 5.2.1.4, the bounds computed in Sections 5.2.1.2 and 5.2.1.3 are employed to compute the worst-case latency of a *read* command (regardless of its classification). As it will become clear, the bound simply selects the largest latency between the *too-early* and *too-late* cases.

For the sake of reading guidance, the information of the previous paragraph is summarized in Table.

Table 5.2.: Structure of the computation of the worst-case latency of *read* commands.

Section	Description
5.2.1.1	Assisting lemma and initial considerations.
5.2.1.2	Worst-case latency of *too-early read* commands.
5.2.1.3	Worst-case latency of *too-late read* commands.
5.2.1.4	Worst-case latency of *read* commands (regardless of classification).

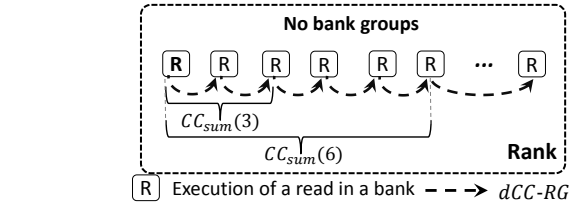
5.2.1.1. Assisting Lemma and Other Initial Considerations

Before the start of the discussion about the worst-case latency of CAS commands, Lemma 5.1, is stated and proven.

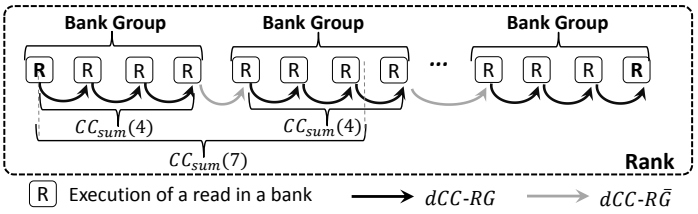
Lemma 5.1 *Given a sequence of $n \leq nB$ CAS commands of the same type that are consecutively executed in different banks of the same rank, the maximum timing interval between the execution of the first and of the last command of such sequence is given by Eq. 5.1.*

$$CC_{sum}(n) = \begin{cases} \textbf{\textit{if}} \;\; nG = 1 \;\textbf{\textit{then:}} & (n-1) \cdot dCC\text{-}R \\ \textbf{\textit{else then:}} & (n-1) \cdot dCC\text{-}RG + \left\lfloor \frac{(n-1)}{\left(\frac{nB}{nG}\right)} \right\rfloor \cdot (dCC\text{-}R\overline{G} - dCC\text{-}RG) \end{cases} \quad (5.1)$$

Proof: The lemma is proven by construction using Figs. 5.1a and 5.1b, which refer respectively to the case in which $nG = 1$ and the case in which $nG \geq 2$. In the figures, latencies are represented using directed edges that connect two commands executed consecutively. Examples of the latencies accounted by CC_{sum} for arbitrary inputs are depicted at the bottom of the figures.



(a) In systems with $nG = 1$.



(b) In systems with $nG \geq 2$.

Figure 5.1.: Graphical depiction of the CC_{sum} function.

The computation of CC_{sum} in systems with $nG = 1$ is simple, as only one type of edges must be accounted (see Fig. 5.1a). In such case, the number of accounted $dCC\text{-}R$ edges is equal to the number of commands in the sequence subtracted by one. For instance, $CC_{sum}(6) = 5 \cdot dCC\text{-}R$.

The computation of CC_{sum} in systems with $nG \geq 2$ is slightly more complex, as it demands taking into account *group interleaving*. More specifically, consecutive commands executed in the same *bank group* take longer to execute ($dCC\text{-}RG > dCC\text{-}R\overline{G}$). Hence, in order to compute a safe bound, the lemma needs to assume that the insertion pattern of interfering CAS commands minimizes the possibility of performing *group interleaving* (see Section 4.3.1). Thinking in graphical terms (see Fig. 5.1b), CC_{sum} must reflect scenarios in which number of solid black edges ($dCC\text{-}RG$) is maximized and the number of solid gray arrows is minimized. Notice that the pattern assumed in Fig. 5.1b clearly

fulfills such goal, as more solid black edges would only be possible if more than one CAS command could be executed in the same bank (which is not addressed by the lemma).

Hence, the discussion now focuses on proving that the computation of CC_{sum} in systems with $nG \geq 2$ accurately describes such pattern. For such purpose, notice that the expression that computes CC_{sum} in systems with $nG \geq 2$ has two terms: the first one assumes that the latency between any two consecutive CAS commands is *dCC-RG* (solid black edges), which is overly conservative. The second term corrects such overly conservative assumption.

More specifically, if n is larger than the number of banks per *bank group* (which is given by nB/nG), at least one pair of consecutive commands will cross the *bank group* boundary. The second term of the equation simply identifies how many command pairs fall into such category and then replaces occurrences of *dCC-RG* by *dCC-*$R\overline{G}$ accordingly. The -1 in the upper part of the fraction inside the *floor* function is necessary because *dCC-*$R\overline{G}$ only occurs if n is larger (and not larger or equal) than the number of banks per *bank group*. This concludes the proof. □

Other initial considerations are now stated. In order to understand the analysis, it is important to notice that the turnaround from read to write is the same regardless of *group interleaving* in systems in which nG is larger than 1. Simply put, *dRW-RG* = *dRW-*$R\overline{G}$ and, hence, the analysis refers to such constraint simply as *dRW-R*. However, this is not the case for the turnaround from write to read. More specifically, *dRW-RG* > *dRW-*$R\overline{G}$. Hence, the equations in the analysis always employ *dWR-RG*.

Finally, it is **very important** to notice that the equations presented in this chapter compute upper bounds on latency and are not a cycle-by-cycle accurate description of the worst-case. More specifically, the computed bounds are larger than the actual worst-case latencies (instead of being larger or equal). This is because the interactions between SDRAM commands get quite complex (especially for DDR4), which forces the equations to make different conservative assumptions that are sometimes mutually exclusive. A good example of such characteristic of the analysis can be observed comparing Lemma 5.2 with Fig. 5.3.

5.2.1.2. Worst-case Latency of Too-Early read Commands

In order to compute the worst-case latency of *too-early read* commands, it is useful to firstly compute the worst-case latency of a scheduling round. The worst-case latency of a scheduling round refers to the largest timing interval between the execution of the first and the last CAS commands executed in the round. From the perspective of the worst-case, two scenarios are considered: WR-rounds, i.e. rounds that started with a W-sweep and end with a R-Sweep, and RW-rounds, which represent the opposite situation.

The remaining of this section firstly states and proves lemmas that compute the worst-case latency of WR- and RW-rounds. Then, it computes the worst-case latency of *too-early read* commands. In order to provide reading guidance, the contents of the this paragraph are summarized in Table 5.3.

Before the computation of the worst-case latency of scheduling rounds is presented,

Table 5.3.: Structure of the proof of the worst-case latency of *too-early read* commands.

Statement	Description
Lemma 5.2	Worst-case latency of WR-round.
Lemma 5.3	Worst-case latency of RW-round.
Lemma 5.4	Worst-case latency of *too-early read* commands.

two observations are made: (1) In order for the latency of a round to be maximized, each command register must provide a CAS command. (2) Rounds in which one of the sweep operations only contains empty visitations, i.e. a purely R-round or a purely W-round, do not experience a data bus turnaround and have less rank switches. Hence, they inevitably lead to an improvement in the latency.

As a consequence, the analysis focuses on rounds in which all command registers provide a CAS command and in which at least one command from each type (*read* or *write*) is executed.

Lemma 5.2 *The worst-case latency of a WR-round is given by Eq. 5.2.*

$$L^{WR\text{-}round} = ccds + switches \tag{5.2}$$

$$\tag{5.3}$$

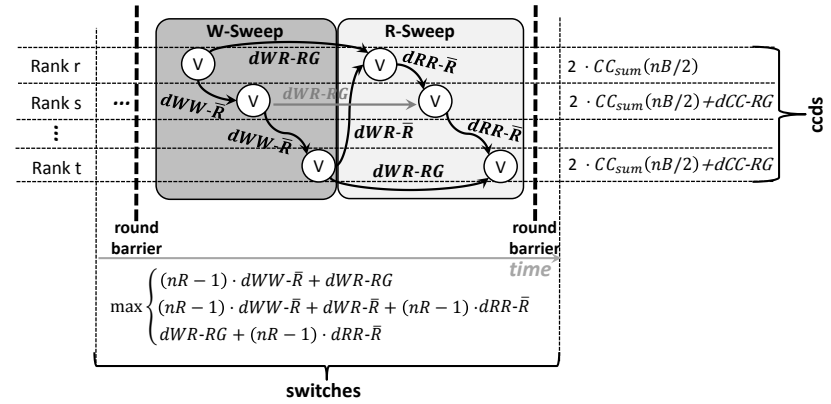
where:

$$ccds = nR \cdot 2 \cdot CC_{sum}(nB/2) + (nR-1) \cdot dCC\text{-}RG \tag{5.4}$$

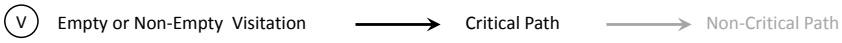
$$switches = \max \begin{cases} (nR-1) \cdot dWW\text{-}\overline{R} + dWR\text{-}RG \\ (nR-1) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} + (nR-1) \cdot dRR\text{-}\overline{R} \\ dWR\text{-}RG + (nR-1) \cdot dRR\text{-}\overline{R} \end{cases} \tag{5.5}$$

Proof: The proof comes from the pattern depicted in Fig. 5.2. The figure depicts the execution of a round in terms of sweeps and rank visitations. Rank visitations contribute to the latency of the scheduling round with dCC latencies, which are accounted for using invocations of the CC_{sum} function. In Eq. 5.2, the dCC latencies are computed within a component called *ccds*, which is depicted in the right portion of Fig. 5.2. Between rank visitations, however, data bus turnarounds and rank switches occur, which also contribute to the latency of the scheduling round. The turnaround and rank switching latencies are computed within a component called *switches*, which is depicted at the bottom of Fig. 5.2.

The component that accounts for dCC latencies is firstly discussed. Such component is computed according to Eq. 5.4, which has two terms. This proof firstly discusses the first term, i.e. $nR \cdot 2 \cdot CC_{sum}(nB/2)$. Textually described, the term assumes a WR-round in which no empty-visitations happen, a scenario depicted in Fig. 5.3a. In such case, each rank contributes with $nB-2$ occurrences of dCC latencies. In systems with nG=1, such $nB-2$ occurrences of dCC latencies could be computed by invoking $CC_{sum}(nB-1)$



(a) In terms of a pattern.



(b) Graphical notation.

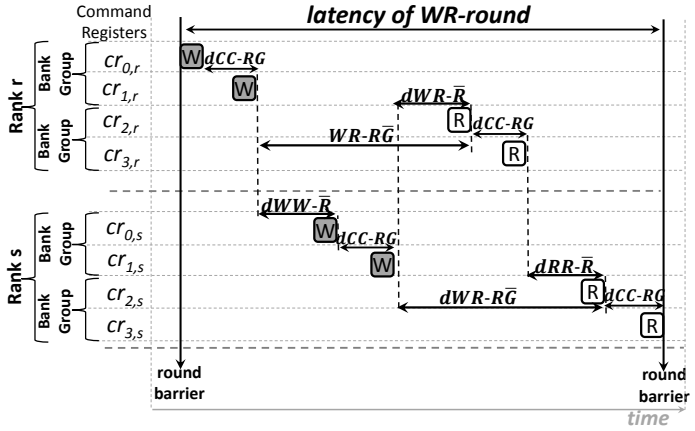
Figure 5.2.: Pattern for worst-case latency of WR-round.

(which would yield the same result as $2 \cdot CC_{sum}(nB/2)$).

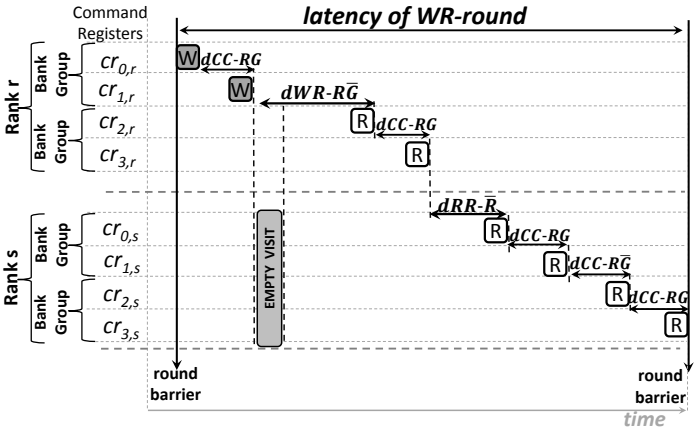
However, so that the Lemma also covers systems with nG larger than 1, the term computes the number of *dCC* latencies experienced in each rank as $2 \cdot CC_{sum}(nB/2)$. To understand the reason, compare $CC_{sum}(7)$ with $2 \cdot CC_{sum}(4)$ in Fig. 5.1b. Notice that the computation of $2 \cdot CC_{sum}(4)$ amounts to $6 \cdot dCC\text{-}RG$, which is larger than $CC_{sum}(7) = 5 \cdot dCC\text{-}RG + dCC\text{-}R\overline{G}$. Notice also that a further partitioning of the invocation of the CC_{sum} function, i.e. $4 \cdot CC_{sum}(nB/4)$, does not make sense, as there are only two sweep operations in each scheduling round. Hence, the first term of the *ccds* component is conservative.

This leads to the discussion of the second term, i.e. $(nR - 1) \cdot dCC\text{-}RG$. The second term covers scenarios in which some of the ranks suffer empty visitations (either in the R- or the W-Sweep). Examples of such scenarios are available in Figs. 5.3b and 5.3c, respectively. The reason such scenario needs to be considered is because if an empty visitation happens, then a data bus turnaround is fully experienced instead of running concurrently with rank switches. Hence, if the rank switching overhead is very small, experiencing a full data bus turnaround leads to a larger latency. That being established, notice that if an empty visitation happens, the rank that suffers such empty visitation sees one less rank switch and one extra *dCC* latency, which is precisely the reason for the second term. Notice also that as the *ccds* component simply sums both terms, its value constitutes an upper-bound on the sum of *dCC* latencies in any WR-round.

Let the proof now address the component that accounts for rank switches and data bus turnarounds. The challenge in the *switches* component is the fact that turnarounds and rank switches can potentially run concurrently. To properly capture the worst-case, Eq. 5.5 simply enumerates the *corner cases*, which are depicted in the figure with black



(a) Scenario 1.

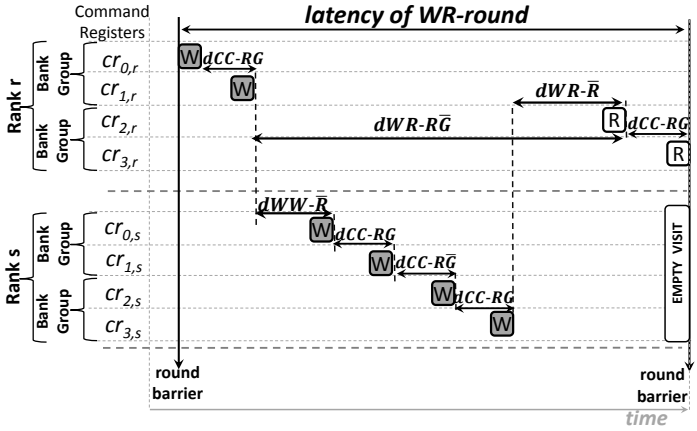


(b) Scenario 2.

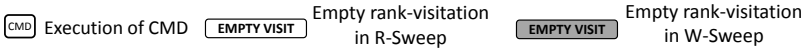
Figure 5.3.: Scenarios that possibly lead to the worst-case latency of a WR-round (continues in next page).

arrows (the so called critical path).

Notice that the figure also includes a gray arrow (to represent a non-critical path). More specifically, such gray arrow refers to a $dWR\text{-}RG$ latency experienced in rank s. The latency incurred by a round that follows a path that goes through such arrow lies between the computed corner cases.



(c) Scenario 3.



(d) Graphical notation.

Figure 5.3.: Scenarios that possibly lead to the worst-case latency of WR-round. Notice that in (a), (b) and (c) constructing a scenario in which dWR-RG is experienced would lead to the replacement of a dCC-RG latency by dCC-$R\overline{G}$ (which would have a positive effect on latency). To be conservative, Lemma 5.2 assumes that it is possible for dWR-RG to be experienced without such replacement. This refers precisely to the last observation made in Section 5.2.1.1.

Finally, notice that in single-rank systems ($nR = 1$), in which any constraint with the $\overline{R}$ modifier is present amounts to zero, Eq. 5.2 yields $2 \cdot CC\_sum(nB/2) + dWR$-$RG$. In plain English, this means a total of $nB - 2$ occurrences of dCC latencies and a single turnaround from write to read.

□

Lemma 5.3 *The worst-case latency of a RW-round is given by Eq. 5.6.*

$$L^{RW\text{-}round} = ccds + switches \tag{5.6}$$

(5.7)

where:

$$ccds = nR \cdot 2 \cdot CC_{sum}(nB/2) + (nR - 1) \cdot dCC\text{-}RG \tag{5.8}$$

$$switches = \max \begin{cases} (nR-1) \cdot dRR\text{-}\overline{R} + dRW\text{-}R \\ (nR-1) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} + (nR-1) \cdot dWW\text{-}\overline{R} \\ dRW\text{-}R + (nR-1) \cdot dWW\text{-}\overline{R} \end{cases} \tag{5.9}$$

Proof: The proof follows exactly the same reasoning than the one employed in the proof for Lemma 5.2. Hence, a discussion is omitted. However, for the comfort of the reader, the pattern used to compute Eq. 5.6 is provided in Fig. 5.4.

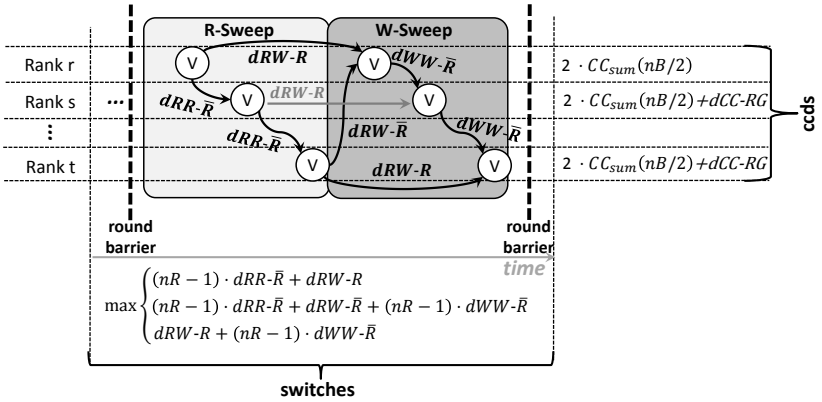


Figure 5.4.: Worst-case latency of RW-round. The same graphical notation as from 5.4 is employed.

□

Lemma 5.4 *The worst-case latency of a* too-early read *is given by Eq. 5.10 if it succeeds a CAS command (SC) in cr u.a. and by Eq. 5.11 if it succeeds a non-CAS command (SNC) in cr u.a..*

$$L^{R}_{SC} = \max \begin{cases} L^{RW\text{-}Round} + WWtrans + L^{WR\text{-}round} - t_{DELAY\_RR}(\text{“}SC\text{”}) \\ L^{WR\text{-}Round} - dCC\text{-}R\overline{G} + RWtrans + L^{WR\text{-}round} - t_{DELAY\_WR}(\text{“}SC\text{”}) \end{cases} \tag{5.10}$$

$$L^{R}_{SNC} = \max \begin{cases} L^{RW\text{-}Round} + WWtrans + L^{WR\text{-}round} - t_{DELAY\_RR}(\text{“}SNC\text{”}) \\ L^{WR\text{-}Round} - dCC\text{-}R\overline{G} + RWtrans + L^{WR\text{-}round} - t_{DELAY\_WR}(\text{“}SNC\text{”}) \end{cases} \tag{5.11}$$

where:

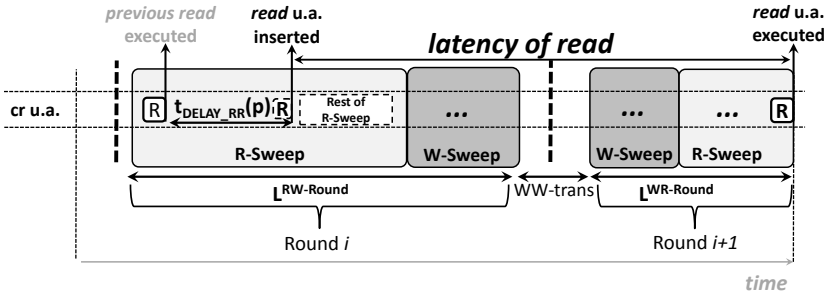
$$t_{DELAY\_WR}(p) = \begin{cases} \texttt{if p = "SC" then:} & dWD + t_{BURST} \\ \texttt{else then:} & dWD + t_{BURST} + dPA\text{-}RGB + dAR\text{-}RGB \end{cases} \tag{5.12}$$

$$t_{DELAY\_RR}(p) = \begin{cases} \texttt{if p = "SC" then:} & dRD + t_{BURST} \\ \texttt{else then:} & dRD + t_{BURST} + dPA\text{-}RGB + dAR\text{-}RGB \end{cases} \tag{5.13}$$

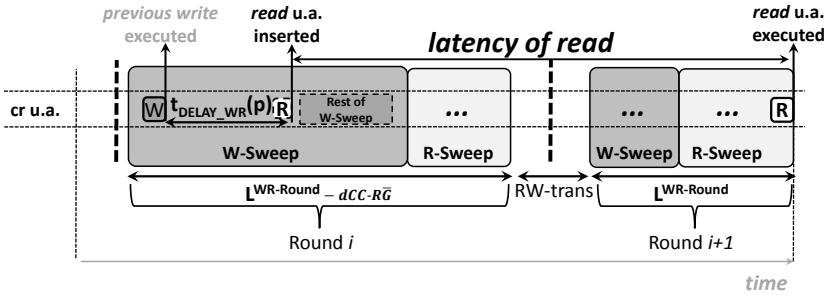
$$WWtrans = \max\{dCC\text{-}RG, dWW\text{-}\overline{R}\} \tag{5.14}$$

$$RWtrans = \max\{dRW\text{-}R, dRW\text{-}\overline{R}\} \tag{5.15}$$

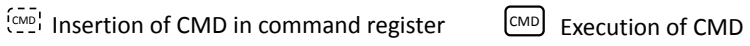
Proof: In order for the latency of a *too-early read* to be computed, the analysis must make the following assumptions: (1) the *read* u.a. is inserted as close as possible to the execution of the previous CAS command that occupied the *cr u.a.*. (2) The previous CAS command that occupied *cr u.a.* was the first CAS command to be executed in its scheduling round. (3) The *read* u.a. is the last CAS command to be executed in its scheduling round.



(a) *Read* u.a. follows a *read* in *cr u.a.*.



(b) *Read* u.a. follows a *write* in *cr u.a.*.



(c) Graphical notation.

Figure 5.5.: Worst-case latency of *too-early read* command. Notice that the *read* u.a. is inserted into *cr u.a.* in round i. However, it is postponed until round $i+1$ because it arrived *too-early*.

The assumptions enforce that the *read* suffers maximum blocking in the round in which it is inserted into *cr u.a.*. Moreover, they also enforce that the *read* suffers maximum blocking in the consecutive round (in which the *read* u.a. is actually executed). That being established, there are two scenarios that need to be considered: either the previous CAS that occupied *cr u.a.* was a *read*, which is depicted in Fig. 5.5a, or the previous CAS that occupied *cr u.a.* was a *write*, which is depicted in Fig. 5.5b. Notice that the example from Fig. 5.5b conservatively assumes that one *write* command was inserted *too-late* during round i, which forces round $i+1$ to start with a W-Sweep. (Hence, a $dCC\text{-}R\overline{G}$ latency is discounted from $L^{WR-round}$.) If round $i+1$ started with a R-sweep, it would benefit the *read* u.a..

Eqs. 5.10 and 5.11 simply reflect the patterns depicted in Figs. 5.5a and 5.5b. In the equations, the terms $WWtrans$ and $RWtrans$ refer to the transition time between consecutive scheduling rounds. Moreover, the minimum distance between the execution of the previous CAS that occupied *cr u.a.* and the insertion of the *read* u.a. into *cr u.a.* is computed with the $t_{DELAY\_RR}()$ and $t_{DELAY\_WR}()$ functions.

Notice that the delay between consecutive CAS commands improves the latency of

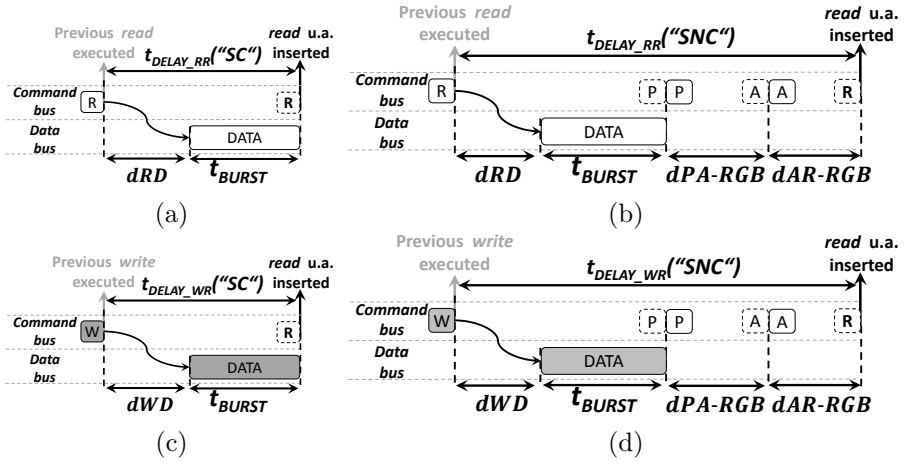


Figure 5.6.: Delays between the execution of the previous CAS command that occupied *cr u.a.* and the insertion of the *read* u.a. into the *cr u.a.*.

the *too-early read* u.a.. Hence, its value must be an actual lower-bound (i.e. as small as possible). For that purpose, the functions rely on the assumption of a timing compositional processor, which only issues a request after the previous request from the same requestor has been served. To clarify why the functions are a lower-bound, the cases they cover are depicted in Fig. 5.6.

For instance, if the *read* u.a. succeeds a non-CAS command, and the previous CAS that occupied *cr u.a.* was also a *read*, the delay between the execution of the previous *read* and the insertion of the *read* u.a. is at least $t_{DELAY\_RR}("SNC")$ cycles. Notice that this constitutes a lower-bound because it only considers exclusively intra-bank constraints and the time required to perform a data transfer.

□

5.2.1.3. Worst-case Latency of Too-Late read Commands

This section computes the worst-case latency of *too-late read* commands. If a command is *too-late*, then the recent history of the *cr u.a.* does not need to be taken into account. Hence, the distinction between the SC and SNC cases is not necessary.

The discussion about the worst-case latency of *too-late reads* is structured into three parts. Firstly, Lemma 5.5 computes the worst-case latency of *too-late reads* for single-rank systems. Then, in Lemma 5.6 does so for multi-rank systems. Finally, Theorem 5.8 binds Lemmas 5.5 and 5.6 in a single equation, so that further steps of the analysis can be performed without addressing an specific value of nR. For the sake of clarity, the contents of this paragraph are summarized in Table 5.4. Moreover, the lemmas and the theorem are available below.

Lemma 5.5 *The worst-case latency of a* too-late read *command in a single-rank system*

Table 5.4.: Structure of the proof of the worst-case latency of *too-late read* commands.

Statement	Computed Value	Description
Lemma 5.5	$L_{SR}^{too\text{-}late\ R}$	Worst-case latency of *read* command in single-rank system.
Lemma 5.6	$L_{MR}^{too\text{-}late\ R}$	Worst-case latency of *read* command in multi-rank system.
Theorem 5.8	$L^{too\text{-}late\ R}$	Convenient way to refer to the latency of a *read* command (regardless of rank setup).

($nR = 1$) is calculated with Eq. 5.16.

$$L_{SR}^{\text{too-late } R} = dRW\text{-}R + 2 \cdot CC_{sum}(nB/2) + dWR\text{-}RG \tag{5.16}$$

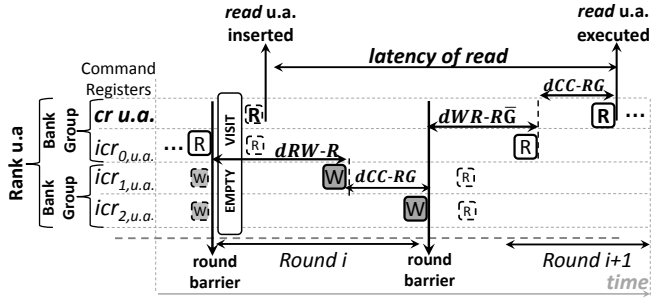
Proof: The lemma is proven using the following properties of the scheduling performed by the controller: (1) In a rank visitation during a sweep operation, commands that are not masked are arbitrated using purely FCFS order (Section 4.3.1). (2) When a *read* is inserted *too-late* during a round that starts in a R-Sweep and ends with a W-Sweep, Algorithm 1 enforces that the next scheduling round starts with R-Sweep. And (3), in a single-rank system, it is simply not possible for a *read* command to be *too-late* for a round that ends in a R-Sweep. That is because the decision to end such R-Sweep would mark the end of the round. *Read* commands inserted immediately after such decision (which initiates a new round) fall into the *just-in-time* category[2].

Because of properties (1) and (2), the *read* u.a. can be blocked at most once by each of the interfering command registers. Moreover, the blocking caused by each interfering register itself can be either a *dCC-R* latency or a turnaround. As turnarounds are always larger than *dCC-R* latencies, we have to assume they happen as often as possible (once in the beginning of the round and once in the middle) when computing worst-case bounds.

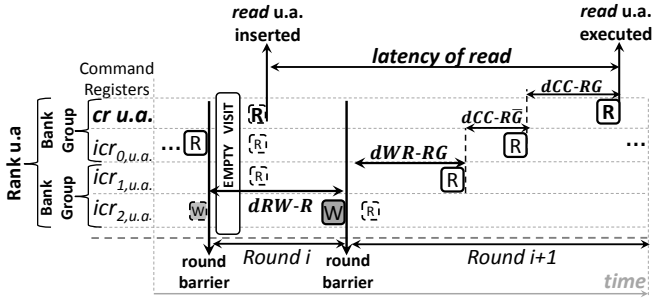
Taking the aforementioned observations into account, two examples of scenarios that lead to the worst-case latency of a *too-late read* are depicted in Figs. 5.7a and 5.7b. Notice that in both examples, when round i ends, function SELECT_SWEEP_TYPE_AND_RANK in Algorithm 1 is invoked with the following parameters: (*rank u.a.*,`W`,`True`,`False`). As a consequence, the round $i+1$ starts with a R-Sweep. Moreover, also in both examples, interfering command registers that block the *read* u.a. in round i will not be able to block the *read* u.a. in round $i+1$ because even if they hold *read* commands, such commands will not be older than the *read* u.a.. For instance, the *reads* in $icr_{1,u.a.}$ and $icr_{2,u.a.}$ in Fig. 5.7a are not older than the *read* u.a. because two *writes* from such set of command registers have been executed in round i. However, command registers that do not block the *read* u.a. in round i can do so in round $i+1$, For instance, the *reads* in $icr_{0,u.a.}$ and $icr_{1,u.a.}$ in Fig. 5.7b block the *read* u.a..

Eq. 5.16 simply takes the aforementioned observations into account. Furthermore, it makes a so far not mentioned conservative assumption: that empty-visitation takes no

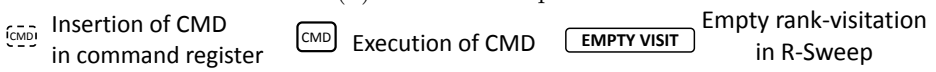
[2] A *just-in-time read* command that fits such description would have exactly the same worst-case latency as a *too-late read*.



(a) First Example.



(b) Second Example.



(c) Graphical Notation.

Figure 5.7.: Examples of worst-case latencies of *too-late reads* commands in single-rank systems.

time (while in actual hardware in takes one cycle). To understand why the assumption is conservative, notice that in the given examples, had the empty visitation took one or more cycles, than the insertion of the *read* u.a. in the *cr* u.a. would be classified as *just-in-time* and not as *too-late*.

Finally, notice that $2 \cdot CC_{sum}(nB/2) = CC_{sum}(nB-1)$ in systems in which $nG = 1$. However, $2 \cdot CC_{sum}(nB/2)$ is larger than $CC_{sum}(nB-1)$ in systems in which $nG > 1$. Hence, as the equation needs to be conservative, the term $2 \cdot CC_{sum}(nB/2)$ is used. □

Lemma 5.6 *The worst-case latency of a* too-late read *command in a multi-rank system ($nR > 1$) is calculated with Eq. 5.17.*

$$L_{MR}^{\text{too-late } R} = ccds + \max\{switches_A, switches_B\} \tag{5.17}$$

where:

$$ccds = 2 \cdot CC_{sum}(nB/2) + dCC\text{-}RG + (nR-1) \cdot 2 \cdot CC_{sum}(nB) \tag{5.18}$$

$$switches_A = dRW\text{-}R + \max \begin{cases} (nR-1) \cdot dWW\text{-}\overline{R} + dWR\text{-}RG \\ (nR-1) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} + (nR-1) \cdot dRR\text{-}\overline{R} \\ dWR\text{-}RG + (nR-1) \cdot dRR\text{-}\overline{R} \end{cases} \tag{5.19}$$

$$switches_B = \max\{dWR\text{-}\overline{R}, (dWR\text{-}RG - dWW\text{-}\overline{R})\} + aux \tag{5.20}$$

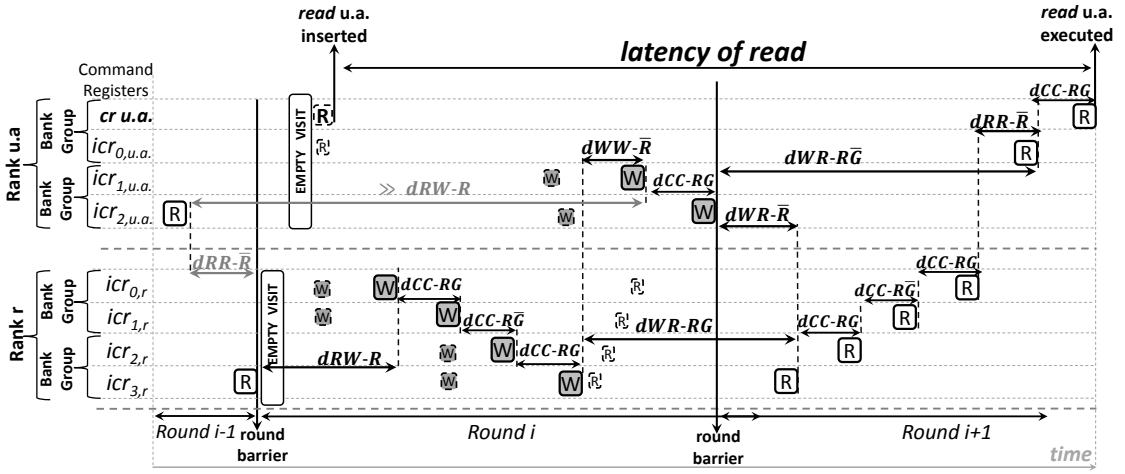
$$aux = \max \begin{cases} (nR-2) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} + dWR\text{-}RG \\ (nR-2) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} + (nR-1) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} \\ (nR-2) \cdot dRR\text{-}\overline{R} + dRW\text{-}R + dWR\text{–}\overline{R} \\ dRW\text{-}R + (nR-2) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} \end{cases} \tag{5.21}$$

Proof: The lemma is again proven using properties of the scheduling performed by the controller. More specifically, three properties, from which the only the first two are equal to the single-rank case: (1) In a rank visitation during a sweep operation, commands that are not masked are arbitrated using purely FCFS order (Section 4.3.1). (2) When a *read* is inserted *too-late* during a round that starts in a R-Sweep and ends with a W-Sweep, Algorithm 1 enforces that the next scheduling round starts with R-Sweep. (3) Unlike the case for a single-rank setup, it is possible for a *read* command to be inserted *too-late* in a scheduling round that ends with a R-Sweep.

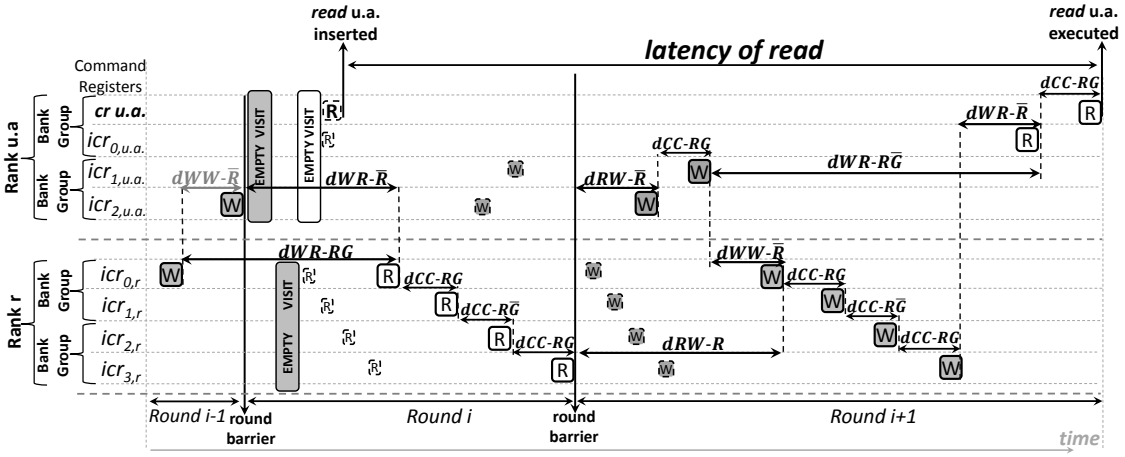
To begin with, notice that properties (1) and (2) enforce that the *read* u.a. is blocked at most once by interfering command registers in the *rank u.a.*. However, because of the two big sweep operations made by the CAS Arbiter, interfering command registers in interfering ranks can block the *read* u.a. up to two times [3].

Moreover, because of property (3), two scenarios in which the *read* u.a. is blocked once by interfering registers in the *rank u.a.* and twice by registers in interfering ranks

[3] The author of this dissertation could not envision scheduling rules to enforce that a *too-late read* command is blocked at most once by interfering command registers in interfering ranks



(a) *Read* u.a. is inserted *too-late* during a round that ends in a W-Sweep.



(b) *Write* u.a. is inserted *too-late* during a round that ends in a R-Sweep.

(c) Graphical notation.

Figure 5.8.: Examples of the two scenarios to be considered when computing the worst-case latency of a *too-late read* command. In the figures, black arrows represent timing intervals that contribute to L^R_{MR}, i.e. such arrows are part of the critical path. Gray arrows represent timing intervals that do not contribute to L^R_{MR}, i.e. they are not part of the critical path. For instance, in Fig. 5.8a, the $dRR\text{-}\overline{R}$ latency in round $i-1$ happens before the insertion of the *read* u.a. and, hence, does not influence its worst-case. Moreover, in the same figure, the arrow labeled as $>> dRW\text{-}R$ (which should be read as significantly larger than $dRW\text{-}R$), can also be ignored because such latency is masked by a read-to-write turnaround experienced in rank r.

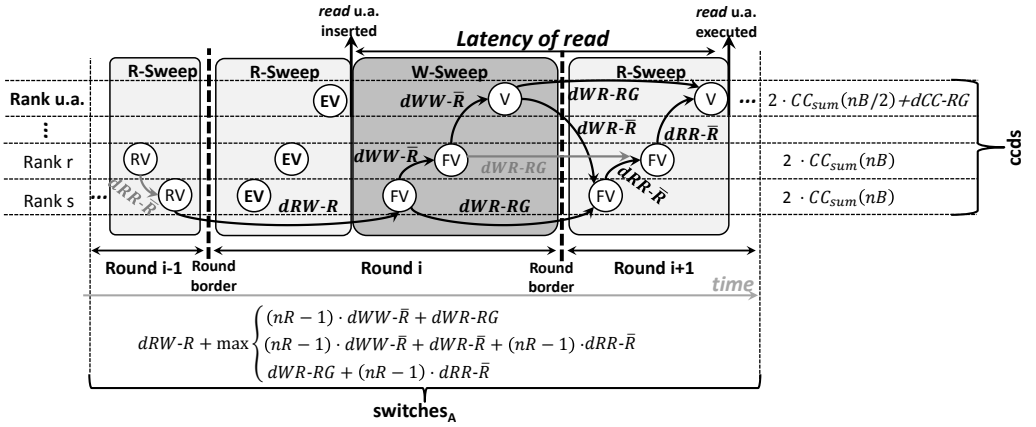
must be considered in order to compute a safe bound: one in which the *read* u.a. is inserted *too-late* in a round that ends with a R-Sweep and one in which the *read* u.a. is inserted *too-late* in a round that ends with a W-Sweep. Examples of both are depicted in Figs. 5.8a and 5.8b, respectively. In the figures, notice that data bus turnarounds run concurrently with rank switches. Notice also that it is not possible to craft a scenario in which the interfering registers in the *rank u.a.* block the *read* u.a. more than once.

Now that the scenarios that can contribute to the worst-case latency have been established, the remaining of the proof will now focus on showing that Eq. 5.17 conservatively describes the scenarios depicted in Figs. 5.8a and 5.8b. For that purpose, it is useful to abstract the latency details that each scenario leads to into patterns. More specifically, Figs. 5.8a and 5.8b can also be represented as the patterns depicted in Figs. 5.9a and Fig. 5.9b. Notice that in each pattern, the latency experienced by the *read* u.a. is broken into two components: one called *ccds*, which accounts for the *dCC* latencies, and one called $switches_x$ (where x can be A or B), which accounts for data bus turnarounds and rank switches, which run concurrently.

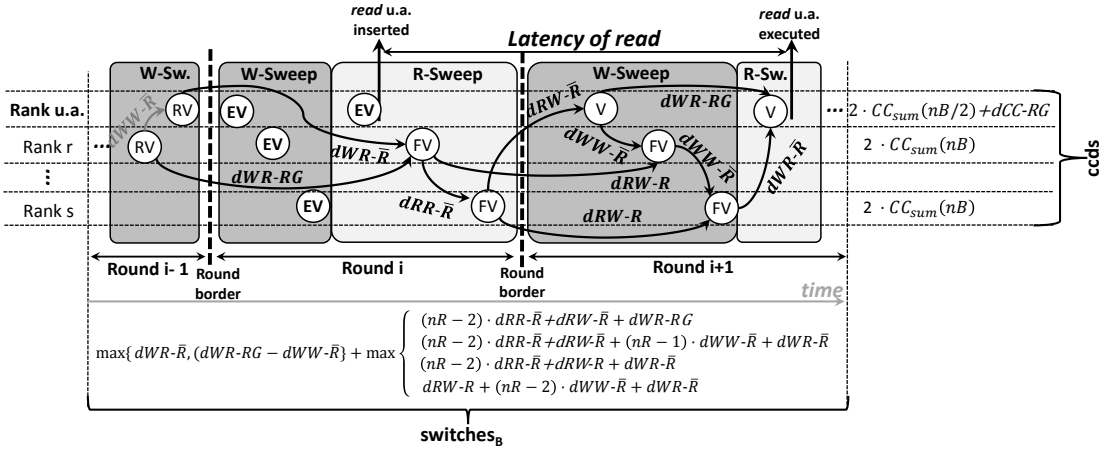
Notice also that, in addition to the already explained empty-visitations, the figures mention full visitations (FV), residual visitations (RV) and visitations (V). A full visitation (FV) refers to a visitation in which one CAS command from each register in the rank being visited is executed. A residual visitation (RV) refers to a non-empty visitation that contributes to what further steps of this proof will call residual latency. Finally, a visitation (V) can refer to empty, full and non-empty visitations. That being said, Eq. 5.17 simply reflects the latencies described in the patterns. More specifically, Eq. 5.17 has two terms: the first one accounts for the *dCC* latencies and the second one for the rank switches (which run concurrently with data bus turnarounds).

The term that accounts for *dCC* latencies is the same regardless of the pattern under consideration and comes directly from the assumption that the *read* u.a. is blocked once by interfering registers in the *rank u.a.* and twice by interfering registers in interfering ranks (as more would not be possible). Notice that the contribution of the rank u.a. to the term that accounts for the *dCC* latencies is $2 \cdot CC_{sum}(nB/2) + dCC\text{-}RG$. In Fig. 5.9a, the *dCC-RG* term is to account for a scenario in which the visitation (V) in the W-Sweep from round i is empty. (Such phenomenon is thoroughly discussed in the computation of the worst-case latency of a WR-round, more specifically in Lemma 5.2). Similarly, in Fig. 5.9b, the *dCC-RG* term is to account for a scenario in which the visitation (V) in the W-Sweep from round $i + 1$ is empty.

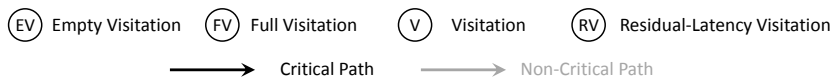
The term that accounts for rank switches and turnarounds uses the max operator to select the largest between $switches_A$ and $switches_B$, which represent the rank switching and turnaround latencies that both patterns under consideration lead to. Eqs. 5.19 and 5.20, which compute $switches_A$ and $switches_B$, consist on enumerating the possible paths between the insertion of the *read* u.a. into its command register and the moment in which the *read* u.a. is executed. For the sake of clarity, this proof now discusses the correctness of $switches_A$, which is computed according to Eq. 5.19. The computation of $switches_A$ consists on a sum of two terms, to which this proof will refer as left and right term, respectively.



(a) The *read* u.a. is inserted *too-late* during a round that ends with a W-Sweep.



(b) The *read* u.a. is inserted *too-late* during a round that ends with a R-Sweep



(c) Graphical notation.

Figure 5.9.: Patterns that describe the worst-case latency of a *too-late read* command. The latency that each pattern induces is broken into two components: one called *ccds*, which accounts for the dCC latencies, and a $switches_x$ (where x can be A or B), which accounts for data bus turnarounds and rank switches, which run concurrently.

The left term (which amounts to $dRW\text{-}R$) computes a residual latency which represents the time interval required for the first CAS command of round i to be executed. Such residual latency is a consequence of the state of SDRAM and, in order for it to be computed, the equation conservatively assumes that empty visitations take no time to be performed (if they did take one or more cycles, the *read* u.a. would be classified as *just-in-time*.

The right term computes the remaining latencies. Notice that it uses the max operator because there are multiple possibilities to go from the FV node in rank s and round i to the V node in the *rank u.a.* in round $i+1$. More specifically, if only the black arrows are considered, there are 3 possible paths, which are simply enumerated in Eq. 5.20. Notice also that the write-to-read turnaround in rank r (the gray arrow labeled as $dWR\text{-}RG$) can be ignored because the path going through it would lead to a latency value between $(nR-1)\cdot dWW\text{-}\overline{R}+dWR\text{-}RG$ and $dWR\text{-}RG+(nR-1)\cdot dRR\text{-}\overline{R}$, which are considered within the max operator.

Finally, the correctness of $switches_B$ is addressed. The computation of $switches_B$ is similar to the one of $switches_A$. The main difference is the fact that, in order to reach a conservative bound, the analysis must assume two empty-visitations in the *rank u.a.* during round i (both in the R- and the W-Sweep). This is necessary because if the *rank u.a.* suffered a non-empty visitation in the R-sweep right before the insertion of the *read* u.a., then there would be no residual latency, which would consequently improve the latency of the *read* u.a.

□

Theorem 5.7 *The worst-case latency of a* too-late read *command is calculated with Eq. 5.23.*

$$L^{\text{too-late } R} = \begin{cases} L_{SR}^{\text{too-late } R} & \textbf{\textit{if}} \quad nR = 1 \\ L_{MR}^{\text{too-late } R} & \textbf{\textit{otherwise}} \end{cases} \tag{5.22}$$

Proof: Eq. 5.23 simply selects L_{SR}^{R} or L_{MR}^{R} depending on the number of ranks of a system (nR). Hence, the proof from Theorem 5.8 comes directly from the proofs from Lemmas 5.5 and 5.6. □

5.2.1.4. Worst-case Latency of CAS Commands

Theorem 5.8 *The worst-case latency of a* read *command is calculated with Eq. 5.23.*

$$L_{SC}^{R} = \max \begin{cases} L_{SC}^{\text{too-early } R} \\ L^{\text{too-late } R} \end{cases} \tag{5.23}$$

$$L_{SNC}^{R} = \max \begin{cases} L_{SNC}^{\text{too-early } R} \\ L^{\text{too-late } R} \end{cases} \tag{5.24}$$

Proof: Eq. 5.23 simply selects L^R_{SR} or L^R_{MR} depending on the number of ranks of a system (nR). Hence, the proof from Theorem 5.8 comes directly from the proofs from Lemmas 5.5 and 5.6. □

5.2.2. Worst-case Latencies of Activate and Precharge Commands

Before the discussion of the worst-case latency of *activate* and *precharge* commands, an assisting lemma, more specifically Lemma 5.9 is stated and proven. Such lemma captures the effect of the lower priority of *activates* and *precharges* with regard to CAS commands (see Section 4.3.3).

Lemma 5.9 *Given a sequence of n* activate *or* precharge *commands that can be immediately executed without violating timing constraints and that can only postpone each other for one cycle due to command bus contention, the maximum timing interval (measured in cycles) required to execute such sequence is given by Eq. 5.25.*

$$\alpha_{PA}(n) = n + \left\lceil \frac{n}{t_{BURST} - 1} \right\rceil \tag{5.25}$$

Proof: Activate and *precharge* commands have lower priority than CAS commands. Hence, to bound the timing interval required to execute them, the analysis has to rely on information about the minimum distance between consecutive CAS commands. More specifically, in single-rank systems, any two consecutive CAS commands must be executed at least t_{BURST} cycles apart (or by even more cycles if a data bus turnaround is required). In multi-rank systems, the same statement remains true if one consider a t_{RTRS} of 4.5 nanoseconds (see Section 2.2 and Table 2.2). Hence, in any interval of t_{BURST} cycles, at least $t_{BURST} - 1$ cycles will be free for the execution of *activates* and *precharges* (see commands between instants t1 and t4 in Fig. 5.10, which assumes $t_{BURST} = 4$). Alternatively, in any interval of $t_{BURST} - 1$ cycles, the stream of *activates* and *precharges* is blocked at most once by a higher-priority CAS (see commands between instants t1 and t3 in Fig. 5.10).

Eq. 5.25 relies on the aforementioned observation and is divided into two terms. The first one simply adds n cycles to the outcome of the function, which refer to the cycles required to execute the n *activate* and/or *precharge* pending commands. The second term computes the blocking caused by higher-priority CAS commands, and comes from the fact that within any window of $t_{BURST} - 1$ cycles, a stream of n pending *activate* or *precharge* commands can be blocked at most once. For the sake of clarity, an example of the outcome of the α_{PA} function is given in Fig. 5.10. This concludes the proof. □

Using Lemma 5.9, the worst-case latency of *activate* commands can be computed according to Theorem 5.10. For ease of comprehension, an example of such latency is depicted in Fig. 5.11. Notice that for the sake of clarity, the figure considers systems with $nG = 1$ and, hence, the G argument is omitted when invoking $dAA\text{-}R\overline{B}$ (Theorem 5.10, however, includes it).

In the figure, the *activate* u.a. is blocked once (more would not be possible) by other *activates* in the *rank u.a.*(if there were *precharges* in the *rank u.a.*, no t_{FAW} or $dAA\text{-}R\overline{B}$

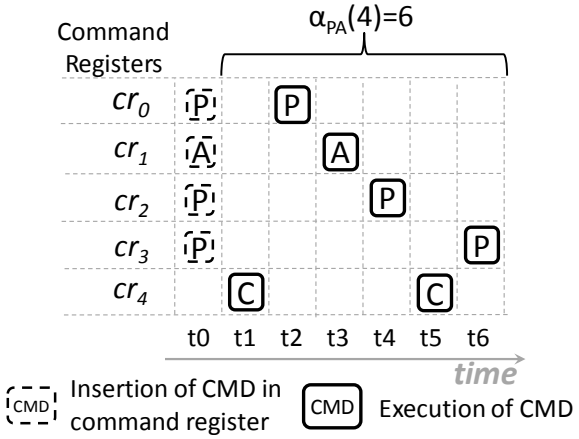


Figure 5.10.: Example of the outcome of the α_{PA} function in a scenario in which $t_{BURST} = 4$ and $nB = 5$.

would be present, thus improving the latency of the *activate* u.a.). More importantly, notice that after a residual latency (consequence of t_{FAW}), whenever an *activate* in the rank u.a. can be immediately executed, such *activate* is blocked by a CAS command and by *activates* from interfering ranks (could have been *precharges*). The blocking through a CAS command, e.g. in instants t_0 and t_1, is a result of CAS commands having higher priority than *activates*. The blocking through *activate* commands in interfering ranks, e.g. in instants $t_0 + 1$ and $t_1 + 1$, is the result of the round-robin arbitration performed in the module-level arbitration of the AP Arbiter (see Section 4.3.2).

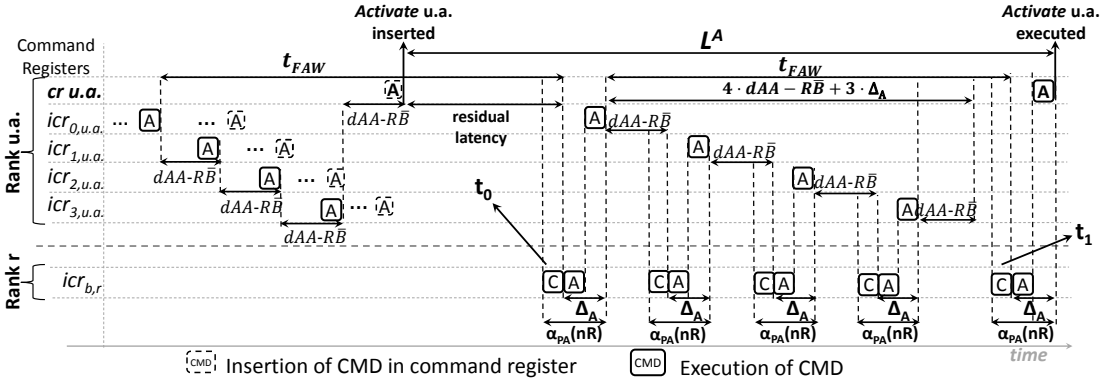


Figure 5.11.: Example of the worst-case latency of an *activate* command in a hypothetical system with nB=5, nG=1 and nR=2. The t_{FAW} constraint is depicted on purpose as significantly larger than $4 \cdot dAA\text{-}R\overline{B}$ in order to highlight its effect. Moreover, the letter C represents a CAS command. Finally, the axis for $icr_{b,r}$ represents all possible registers from rank r.

Theorem 5.10 *The worst-case latency of a* activate *command is calculated using Eq. 5.26.*

$$L^A = (t_{FAW} - (4 \cdot dAA\text{-}R\overline{GB})) + \max \begin{cases} aux, \\ aux + (t_{FAW} - (4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A)) \cdot K \end{cases} \tag{5.26}$$

where

$$aux = (nB - 1) \cdot dAA\text{-}R\overline{GB} + nB \cdot \Delta_A \tag{5.27}$$

$$\Delta_A = \alpha_{PA}(nR) - 1 \tag{5.28}$$

$$K = \left\lfloor \frac{(nB - 1)}{4} \right\rfloor \tag{5.29}$$

Proof: The equation that computes L^A has two main terms. The leftmost term accounts for the residual latency depicted in Fig. 5.11, which comes from the conservative assumption that 4 *activates* are executed as late as possible in the rank u.a..

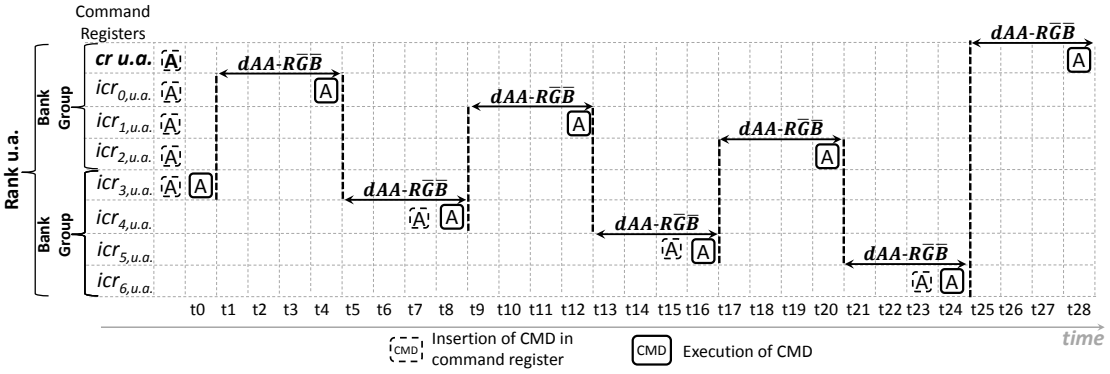
The rightmost term (max operator) accounts for the remaining latencies by selecting the largest value between two expressions. The first one (upper expression) considers that the influence of t_{FAW} is hidden due to inter-rank and CAS interference. More specifically, it considers that $t_{FAW} < 4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A$ (which is not the case in Fig. 5.11). The second one (lower expression) considers exactly the opposite ($t_{FAW} > 4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A$) and simply replaces ($4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A$) by t_{FAW} in each of the K times in which the t_{FAW} constraint is activated.

Finally, the use of the $\overline{G}$ modifier in the equations is discussed. In systems with nG=1, the G modifier is simply ignored. However, in systems with $nG \geq 2$, such modifier is important and the use of $\overline{G}$ in the equations needs to be justified.

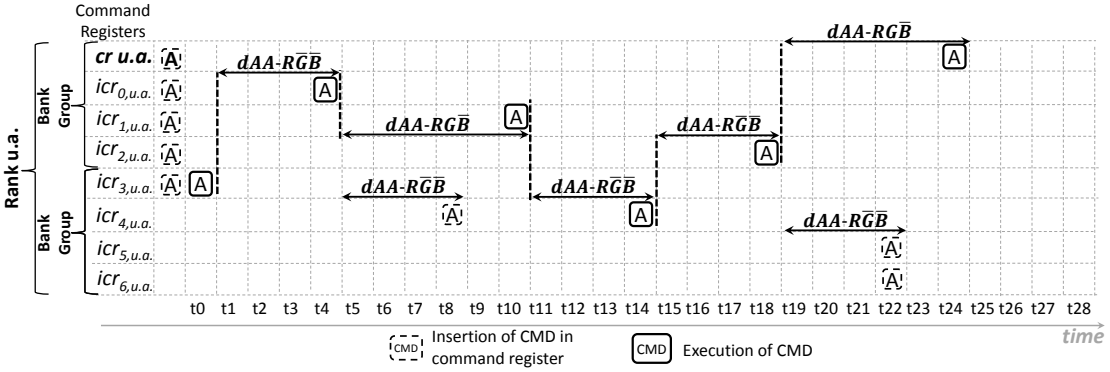
For the residual latency (leftmost term of Eq. 5.26), using $\overline{G}$ obviously increases the outcome of the computation, simply because ($t_{FAW} - 4 \cdot dAA\text{-}R\overline{GB}$) is larger than ($t_{FAW} - 4 \cdot dAA\text{-}RG\overline{B}$). In the expressions inside the max operator, $\overline{G}$ is employed because of the *real-time aware oldest ready* arbitration of the intra-rank portion of the AP Arbiter (see Section 4.3.2). More specifically, $\overline{G}$ is used because if the intra-rank non-t_{FAW} interference experienced by the *activate* u.a. is solely considered, such interference is maximized if two conditions are simultaneously satisfied:

1. firstly, the *activate* u.a. is blocked by $nB - 1$ interfering *activates* (more is not possible).
2. And secondly, all interfering command registers (in the same rank as the *activate* u.a.) suffer insertions of *activates* soon enough for a full *group interleaved* command execution to happen.

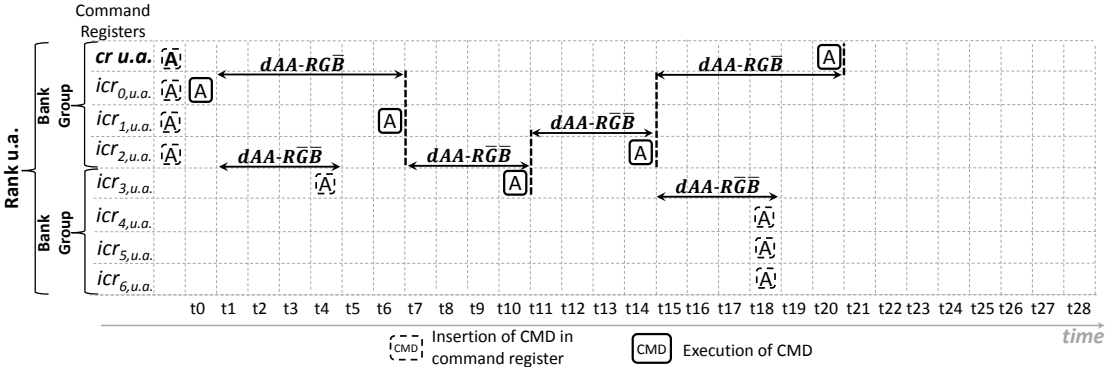
Such scenario is depicted in Fig. 5.12a. Proving that it indeed depicts the maximum interference can be accomplished by observing what happens if the interfering command registers suffer late insertions of *activates*, which in turn prevent a full *group interleaved*



(a) Worst-case interference.



(b) Not the worst-case interference. Notice that in $t9$, an arbiter that purely prioritizes the oldest ready command would execute the *activate* from $icr_{4,u.a.}$, which is not the case for an arbiter that employs *real-time aware oldest ready* arbitration. A similar observation can be made about instant $t23$.



(c) Also not the worst-case interference. Similarly to (b), the *real-time aware oldest ready* arbitration prevents execution from the second bank group in instants $t5$ and $t19$.

Figure 5.12.: Examples of non-t_{FAW} intra-rank interference that an *activate* command can suffer in a system with nR=1, nG=2 and nB=8 and in which $dAA\text{-}R\overline{GB} = 4$ and $dAA\text{-}RG\overline{B} = 6$. For the sake of the examples, t_{FAW} is assumed to be smaller than $4 \cdot dAA\text{-}R\overline{GB}$.

execution pattern. This is depicted in Figs. 5.12b and 5.12c. The remaining of this proof will focus on the former, as it leads to a larger latency than the latter.

Notice that in the figure under consideration, the *activate* u.a. suffers a blocking that amounts to $\left(\frac{nB/nG}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{nB/nG}{2} - 1\right) \cdot dAA\text{-}R\overline{GB}$. The left term of the expression comes from the observation that for every pair of command registers inside the *bank group* from the *cr u.a.*, an interference of $(dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B})$ is observed (e.g. see the two latencies that follow the execution of the *activate* in $icr_{3,u.a.}$ or the two latencies that follow the execution of the *activate* in $icr_{4,u.a.}$). The right term accounts for the latencies between the execution of *activates* in the command register pairs mentioned in the explanation of the left term, e.g. the $dAA\text{-}R\overline{GB}$ between $t11$ and $t14$ in Fig. 5.12b. Notice that, regardless of the number of *bank groups* in the system, a larger blocking would only be possible if the *activates* outside the *bank group* of the *cr* u.a. were inserted earlier (scenario from Fig. 5.12a).

This leads to the last part of the proof. More specifically, let the analysis assume that the scenario depicted in Fig. 5.12b leads to worst-case interference in a system in which the number of banks per *bank group* is equal to 4 ($nB/nG = 4$), which is the case for DDR4 systems. This means that such interference would be larger than the one depicted in Fig. 5.12a and leads to the following inequality:

$$\left(\frac{nB/nG}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{nB/nG}{2} - 1\right) \cdot dAA\text{-}R\overline{GB} > (nB-1) \cdot dAA\text{-}R\overline{GB}$$

$$\left(\frac{4}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{4}{2} - 1\right) \cdot dAA\text{-}R\overline{GB} > (nB-1) \cdot dAA\text{-}R\overline{GB}$$

$$2 \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + 1 \cdot dAA\text{-}R\overline{GB} > (nB-1) \cdot dAA\text{-}R\overline{GB}$$

$$2 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} + 1 \cdot dAA\text{-}R\overline{GB} > (nB-1) \cdot dAA\text{-}R\overline{GB}$$

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > (nB-1) \cdot dAA\text{-}R\overline{GB}$$

In systems with 8 banks per rank, it is possible to further develop the expressions until a false statement is reached:

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > (8-1) \cdot dAA\text{-}R\overline{GB}$$

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > 7 \cdot dAA\text{-}R\overline{GB}$$

$$2 \cdot dAA\text{-}RG\overline{B} > 4 \cdot dAA\text{-}R\overline{GB}$$

$$dAA\text{-}RG\overline{B} > 2 \cdot dAA\text{-}R\overline{GB} \quad \textbf{False!!!}$$

(Notice that $dAA\text{-}RG\overline{B}$ is indeed larger than $dAA\text{-}R\overline{GB}$, but never twice larger, as claimed by the last line of the inequation.)

Similarly, in systems with with 16 banks per rank, the expressions can be developed until a false statement is reached:

$$\begin{aligned} 3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} &> (16-1) \cdot dAA\text{-}R\overline{GB} \\ 3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} &> 15 \cdot dAA\text{-}R\overline{GB} \\ 2 \cdot dAA\text{-}RG\overline{B} &> 12 \cdot dAA\text{-}R\overline{GB} \\ dAA\text{-}RG\overline{B} &> 6 \cdot dAA\text{-}R\overline{GB} \quad \textbf{False!!!} \end{aligned}$$

Consequently, it is possible to affirm that the worst-case interference happens in the scenario from Fig. 5.12a. Finally, it is to be highlighted that in future systems, in which the number of banks per *bank group* might be larger than 4, a proof can be achieved employing a similar strategy. This concludes the proof. □

Finally, the worst-case latency of a *precharge* command is addressed. Such latency is computed according to Theorem 5.11.

Theorem 5.11 *The worst-case latency of a* precharge *command is calculated according to Eq. 5.30.*

$$L^P = \alpha_{PA}(nB \cdot nR) \tag{5.30}$$

Proof: Precharge commands can be executed back-to-back regardless of the rank. Moreover, a *precharge* can be blocked at most once by *precharge* or *activate* commands in interfering banks. Eq. 5.30 reflects the aforementioned observations. Finally, notice that $nB \cdot nR$ (instead of $nB \cdot nR - 1$) is employed as an argument to the $\alpha_{PA}(n)$ function, as one cycle to execute the *precharge* u.a. is also accounted for. □

5.3. Worst-case Latency of a Request

This section discusses the worst-case latency of a SDRAM request. The worst-case latency of a SDRAM request refers to the maximum time interval between its arrival at the SDRAM controller and the end of the corresponding data transfer. Such latency depends on three factors: the worst-case command latencies required to serve the request (which were computed in the previous section), the corresponding intra-bank constraints and the time interval required for the corresponding data transfer to happen. In total, there are four types of request to be considered: read miss (RM), read hit (RH), write miss (WM) and write hit (WH). The words miss and hit are not related to cache and instead refer to whether the request u.a. targets a row currently present in the corresponding *row buffer* or not. If that is the case, only a CAS command is required. If that is not the case, than a P-A-CAS command sequence is required.

That being said, Theorem 5.12 is stated and proven.

Theorem 5.12 *The worst-case latency of a* Read Miss (RM) *and of a* Write Miss (WM) *requests are given by Eqs. 5.31 and 5.32, respectively.*

$$L_{Req}^{RM} = t_{Residual} + L^{P} + L^{A} + L_{SNC}^{R} + dPA\text{-}RGB + dAR\text{-}RGB + dRD + t_{BURST} \tag{5.31}$$

$$L_{Req}^{WM} = t_{Residual} + L^{P} + L^{A} + L_{SNC}^{W} + dPA\text{-}RGB + dAW\text{-}RGB + dWD + t_{BURST} \tag{5.32}$$

$$\tag{5.33}$$

where:

$$t_{Residual} = \begin{cases} residual\_prev\_RH & \texttt{if previous request was RH} \\ residual\_prev\_RM & \texttt{if previous request was RM} \\ residual\_prev\_WH & \texttt{if previous request was WH} \\ residual\_prev\_WM & \texttt{if previous request was WM} \\ 0 & \texttt{if previous request is None} \end{cases} \tag{5.34}$$

$$residual\_prev\_RH = \max \begin{cases} (dRP\text{-}RGB - (dRD + t_{BURST})) \\ 0 \end{cases} \tag{5.35}$$

$$residual\_prev\_RM = \max \begin{cases} (dAP\text{-}RGB - (dAR\text{-}RGB + dRD + t_{BURST})) \\ dRP\text{-}RGB - (dRD + t_{BURST}) \\ 0 \end{cases} \tag{5.36}$$

$$residual\_prev\_WH = \max \begin{cases} (dWP\text{-}RGB - (dWD + t_{BURST})) \\ 0 \end{cases} \tag{5.37}$$

$$residual\_prev\_WM = \max \begin{cases} (dAP\text{-}RGB - (dAW\text{-}RGB + dWD + t_{BURST})) \\ dWP\text{-}RGB - (dWD + t_{BURST}) \\ 0 \end{cases} \tag{5.38}$$

Proof: In order to aid the proof, the latencies that contribute to the worst-case latency of a RM request are depicted in Fig. 5.13. (The case for WM is similar). Notice that between the execution of a command and the insertion of the next command into the command register u.a., the corresponding intra-bank latencies must be respected. For instance, after the execution of a *precharge*, the corresponding bank scheduler has to wait $dPA\text{-}RGB - 1$ cycles before inserting an *activate* into the command register u.a.. (So the *activate* becomes visible for the AP Arbiter $dPA\text{-}RGB$ cycles after the execution of the *precharge*.)

Eqs. 5.31 and 5.32 simply sum worst-case command latencies, the intra-bank latencies and the data transfer duration. However, one characteristic of the equations for RM and WM requests demands a clarification. More specifically, the $t_{Residual}$ term. The $t_{Residual}$ latency is a consequence of intra-bank timing constraints ($dAP\text{-}RGB$, $dRP\text{-}RGB$ and $dWP\text{-}RGB$) that limit how fast a *precharge* can be inserted into the command register u.a.. Such latency depends on the type of the request that preceded the request u.a. and, in order to compute it, the lemma assumes that the request u.a. arrives exactly after the previous request has been served, as depicted in Fig. 5.13

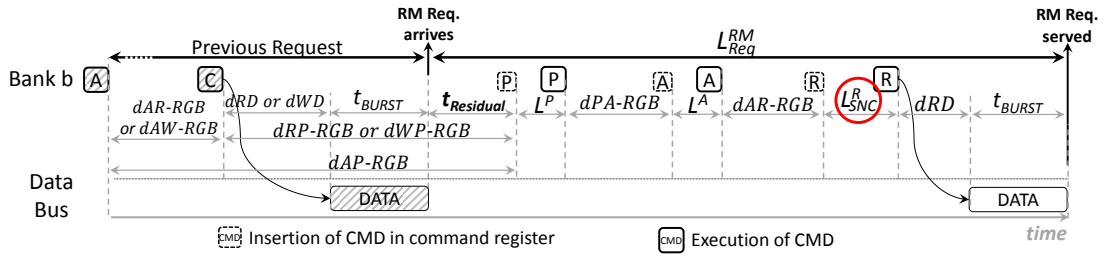


Figure 5.13.: Latency decomposition of a RM request. The figure is not drawn to scale and the employed proportions are chosen solely to properly fit the latency labels. Moreover, the letter C refers to a CAS command. Notice that L^{R}_{SNC} is employed to account for the worst-case latency of the *read* command.

□

Theorem 5.13 *The worst-case latency of a* Read Hit (RH) *and of a* Write Hit (WH) *requests are given by Eqs. 5.39 and 5.40, respectively.*

$$L^{RH}_{Req} = 1 + L^{R}_{SC} + dRD + t_{BURST} \tag{5.39}$$

$$L^{WH}_{Req} = 1 + L^{W}_{SC} + dWD + t_{BURST} \tag{5.40}$$

Proof: The proof comes directly from Fig. 5.14 (the figure depicts the latency of a RH request, the case for WH is similar). Basically, as soon as the request arrives at the controller, the corresponding CAS command is inserted into the *cr u.a.* (notice that the equation accounts for the 1 cycle required for the insertion). After the insertion, the latency of the CAS command is bounded by L^{R}_{SC} (or L^{W}_{SC}). After its execution, the corresponding data transfer will be completed after $dRD + t_{BURST}$ (or $dWD + t_{BURST}$) cycles.

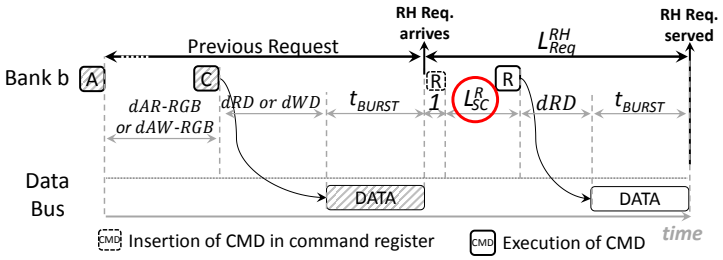


Figure 5.14.: Latency decomposition of a RH request. The letter C refers to a CAS command. Notice that L^{R}_{SC} is employed to account for the worst-case latency of the *read* command.

□

5.4. Worst-case Latency of a Task

The worst-case cumulative SDRAM latency of a task (L_{Task}^{SDRAM}) refers to the the maximum amount of time that a task spends idle while waiting for its memory requests to be served. In order to compute it, this section assumes that the number and the types of requests made by a task are extracted from a trace. This eases the computation of L_{Task}^{SDRAM} because the appropriate value of $t_{Residual}$ can always be selected for RM and WM requests. (If such assumption cannot be made, a static timing analysis tool [15, 107, 124] can be employed to extract the maximum number and type of requests that a task can perform[4]. Moreover, a worst-case bound on the sum of all $t_{Residual}$ latencies must be derived. Such bound can be computed according to [126].)

The worst-case cumulative SDRAM latency of a task is computed according to Corollary 5.14. One important highlight is made about the theorem, though: for the sake of simplicity, it deliberately disregards the effect that refreshes have in L_{Task}^{SDRAM}. This is because, as discussed in [67], the effect of refreshes is negligible in comparison with other command delays, provided that the execution time of the task u.a is not too short. For tasks that fall into the short scenario, a software approach for predictable refreshes is available at [12].

Corollary 5.14 *The worst-case SDRAM Latency of a task whose SDRAM request trace is available is computed using Algorithm 4.*

Proof: The algorithm simply sums the latency of every request in the trace, which comes directly from the assumption of a timing compositional processor that stalls at every request (see Section 5.1). □

Algorithm 4 Computes L_{Task}^{SDRAM}

```
1: // Inputs: N (number of requests) and request_trace (a trace with N requests)
2: function COMPUTE_CUMULATIVE_WC_LATENCY(N, request_trace[N])
3:     L_Task^SDRAM ← 0 ;
4:     previous_request ← None ;
5:     current_request ← None ;
6:     for index ← 0; index < N; index ← index + 1 do
7:         current_request ← request_trace[index] ;
8:         t_Residual ← GET_TRESIDUAL(previous_request) ;
9:         L_Task^SDRAM ← L_Task^SDRAM + GET_REQUEST_LATENCY(t_Residual, current_request) ;
10:        previous_request ← current_request ;
11:    return L_Task^SDRAM ;
```

[4] To be highlighted is that out of the mentioned articles, [15] is the only to focus on the issue of *row buffer* locality. More specifically, [15] is able to compute the number of read miss, read hit, write miss and write hit requests, while articles such as [107] compute the number of read and write requests.

5.5. Considerations About Bank Sharing

If tasks being simultaneously executed in different cores need to exchange data or if the number of SDRAM banks in a system is smaller than the number of requestors, having only *private* banks will not suffice. In such case, one (or more banks) of the SDRAM must be shared (see Section 4.4). This section discusses how to to compute $L_{Task}^{Shared\_SDRAM}$, the worst-case cumulative SDRAM latency of a task that uses a single shared SDRAM bank.

With regard to it, two observations are made: first, an analysis for a scenario in which multiple shared SDRAM banks are employed is similar and, hence, will not be addressed. And second, if a task has access to both a *private* and a *shared* bank, then its total worst-case cumulative SDRAM latency has two components that are independently calculated: L_{Task}^{SDRAM}, which accounts for requests to the *private* bank, and $L_{Task}^{Shared\_SDRAM}$, which accounts for requests to the *shared* bank.

In order to compute the guarantees for a shared bank setup, the assumptions made in Section 5.1 must be updated. More specifically, the 5-th assumption from Section 5.1 is removed. Furthermore, four new assumptions are added into the assumption-set:

1. the SDRAM bank targeted by the request u.a. is shared by a total of N_R requestors, including the requestor u.a..
2. The shared bank is marked as *critical* and, as a consequence, the bank request queue in the controller employs the FCFS policy.
3. Requests for a shared bank always miss at the *row buffer*. (This is conservative, as a request that misses at the *row buffer* always has a larger latency than a request that hits at the *row buffer*.)
4. Each of the N_R requestors is only able to make a SDRAM request after the previous request has been served. If there are *best-effort* applications running in superscalar processors (that tolerate multiple pending requests) among the N_R requestors, then they must be monitored according to what is described in Section 4.4 in order to enforce the assumed behavior.

The remaining of this section firstly states Theorem 5.15, which computes the worst-case latency of read and write requests that target a shared bank. Such latencies are then employed to compute $L_{Task}^{Shared\_SDRAM}$ in Corollary 5.16.

Theorem 5.15 *The worst-case latency of a* read *and of a* write *requests that target a shared bank are given by Eqs. 5.31 and 5.32, respectively.*

$$L_{Req}^{Shared\_R} = aux\_read + (N_R - 1) \cdot \max\{aux\_read, aux\_write\} \tag{5.41}$$

$$L_{Req}^{Shared\_W} = aux\_write + (N_R - 1) \cdot \max\{aux\_read, aux\_write\} \tag{5.42}$$

where:

$$aux\_read = t_{Residual} + L^{P} + L^{A} + L^{R}_{SNC} + dPA\text{-}RGB + dAR\text{-}RGB + dRD + t_{BURST} \quad (5.43)$$

$$aux\_write = t_{Residual} + L^{P} + L^{A} + L^{W}_{SNC} + dPA\text{-}RGB + dAW\text{-}RGB + dWD + t_{BURST} \quad (5.44)$$

$$t_{Residual} = \max \begin{cases} dRP\text{-}RGB - (dRD + t_{BURST}) \\ dAP\text{-}RGB - (dAR\text{-}RGB + dRD + t_{BURST}) \\ dWP\text{-}RGB - (dWD + t_{BURST}) \\ dAP\text{-}RGB - (dAW\text{-}RGB + dWD + t_{BURST}) \\ 0 \end{cases} \quad (5.45)$$

Proof: Firstly, let the proof address the values of aux_read and aux_write. The former computes the worst-case latency of a read request that misses at the *row buffer* while the latter does so for a write request that misses at the *row buffer*. The expressions are very similar to the ones used to compute L^{RM}_{Req} and L^{WM}_{Req} (see Theorem 5.12). The only difference is that in Theorem 5.12, $t_{Residual}$ is defined in terms of the previous request (which can be RM, WM, RH and WH), while in the computation of aux_read and aux_write, $t_{Residual}$ is computed using the max operator to select the largest residual latency among all possible cases. This is done in order to keep it independent from the type of requests issued by interfering requestors. As a consequence, the computation of aux_read and aux_write is conservative.

That being established, the proof now addresses the equation that computes $L^{Shared\_R}_{Req}$. Notice that the equation contains two terms: the left term accounts for the latency of the read request u.a.. The right term accounts for intra-bank interference. More specifically, the second term assumes that each of the other $N_R - 1$ requestors that use the shared bank u.a. also have a pending request. Moreover, each of these interfering requests is assumed to cause as much interference as possible (i.e. the equations use the max operator to select between read and write requests). Notice that because of the 4-th assumption presented earlier in this section, it is not possible for the request u.a. to be blocked more than once by the $N_R - 1$ interfering requestors. Hence, as the computation of aux_read and aux_write is conservative, so is the computation of $L^{R\_Shared}_{Req}$.

The reasoning for $L^{Shared\_W}_{Req}$ is similar and, hence, a discussion is omitted. This concludes the proof. □

Corollary 5.16 *The worst-case cumulative SDRAM latency of a task that uses a single shared SDRAM bank is computed using Equation 5.46, where N_{reads} and N_{writes} refer to the number of read and write requests made by the task, respectively.*

$$L^{Shared\_SDRAM}_{Task} = N_{reads} \cdot L^{R\_Shared}_{Req} + N_{writes} \cdot L^{W\_Shared}_{Req}. \quad (5.46)$$

Proof: Eq. 5.46 simply multiplies the requests by their corresponding worst-case latency. □

Finally, two further remarks are made. Firstly, notice that no *refresh* is accounted for in Corollary 5.16. However, the same considerations about *refreshes* made in the previous section regarding the statement of Corollary 5.14 apply.

Secondly and most importantly, notice that Theorem 5.15 and Corollary 5.16 assume that every request made by the task u.a. always experiences maximum blocking due to requests from interfering requestors. Although this strategy simplifies the computation, it is certainly overly conservative. To understand why, consider a scenario in which the task u.a. shares a bank with a single interfering requestor which is also *critical*. As both requestors are *critical*, their behavior is well known, i.e. their memory access patterns are well known.

Hence, if it is known that the task u.a. issues at most 5000 requests in any 10000 ms window, and if it is also known that the interfering requestor makes at most 2000 requests in any 10000 ms window, the assumption made by Theorem 5.15 and Corollary 5.16 is overly conservative, as at most 2000 requests of the task u.a. will suffer intra-bank interference. Consequently, a timing bound that takes into account such effect, such as the ones computed by [107], would be tighter. Notice, however, that because interfering requestors are assumed to tolerate at most a single pending memory request, the difference in tightness between the two approaches is not large. (In the example under consideration, if the interfering requestor could tolerate up to 10 pending memory requests, Theorem 5.15 and Corollary 5.16 would assume that each request from the task u.a. would be blocked by 10 interfering requests instead of one.)

5.6. Summary

This chapter presents a timing analysis of the DDR SDRAM controller proposed in Chapter 4. Similarly to other *open-row* controllers in the literature, guarantees are computed in terms of all requests performed by a *critical* task. More specifically, a worst-case cumulative SDRAM latency is computed, i.e. the maximum amount of time that a *critical* task spends idle while waiting for its memory requests to be served (see Fig. 3.7 in Section 3.2.2). In combination with a *private*-bank assumption, the computation of a cumulative guarantee captures the effect of *row buffer* locality.

The timing analysis, which is performed in terms of minimum distances between consecutive commands represented using the notation discussed in Section 2.3, is broken into three parts: the first part computes the worst-case latency of individual commands. The second part uses the worst-case latencies of commands computed in the first part in order to compute the worst-case latency of requests. The third part combines the worst-case latency of requests in order to compute the worst-case cumulative SDRAM latency.

Bank sharing, which is necessary to implement data sharing or if the number of requestors in a system is larger than the number of SDRAM banks, is addressed separately in the end of the chapter. Requests that target the shared bank(s) are always assumed to miss at the *row buffer*.

6. Evaluation

This chapter evaluates the controller proposed in Chapter 4 and analyzed in Chapter 5. The evaluation is based on SDRAM request traces, which are used not only for the computation of analytical bounds but also as stimuli for cycle-accurate simulators of both the controller proposed in this dissertation and competing controllers.

The evaluation is structured as follows: firstly, Section 6.1 discusses the generation of the request traces and cycle-accurate simulations. Secondly, Section 6.2 assesses the influence of bank partitioning in timing isolation. Thirdly, Section 6.3 presents a comparison of the controller proposed in this dissertation with other SDRAM controllers proposed in the literature. Then, Section 6.4 compares worst-case performance trends in SDRAM modules from different DDR generations and speed bins. Finally, Section 6.5 presents a summary of the evaluation.

6.1. Application Request Traces, Trace Summarization and Cycle-Accurate Simulations

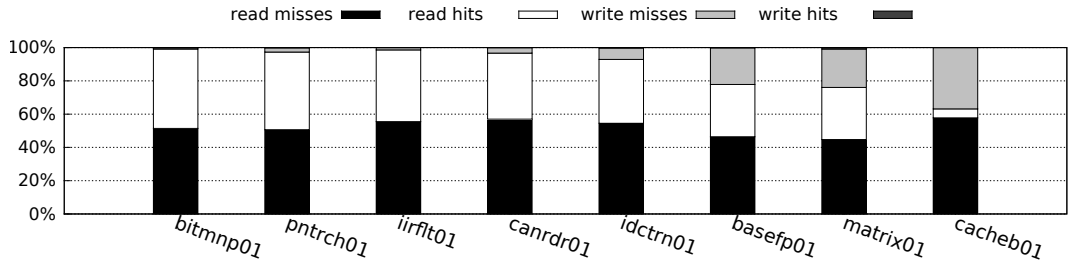
In order to collect traces, a set of 16 applications, 8 from EEMBC [20] and 8 from Mibench [44], are executed in Gem5 [14] with a 1.1 GHz scalar ARM processor with 64-KB of L1 cache (split evenly between instructions and data). The cache line size is 64 bytes, which matches the access granularity of a SDRAM module with a 64-bit data bus, a setup assumed for all experiments in this chapter. Moreover, the cache employs the write-back policy.

The profile of the applications, i.e. the proportion of request types observed in the collected traces, are depicted in Figs. 6.1a and 6.1b for EEMBC and Mibench, respectively. Notice that the profiles are quite different. More specifically, some provide a high number of requests that hit in the *row buffer* and some do not. Similarly, some provide a large number of write requests and some do not.

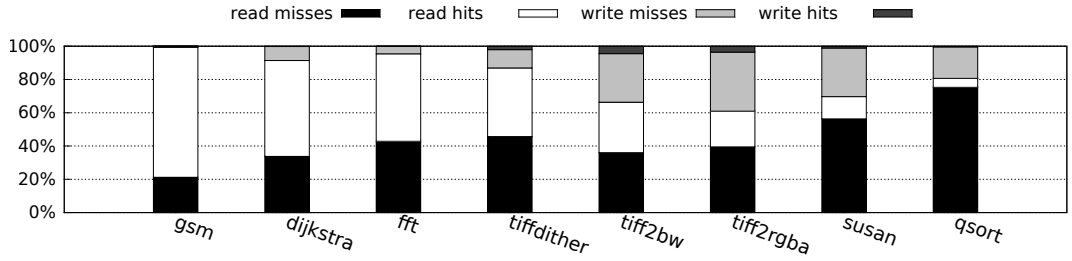
Because the number of requests of each collected trace varies drastically (from a couple thousand to hundreds of thousands), traces are *summarized*. More specifically, artificial traces are generated, each containing 5000 requests, but respecting the proportions depicted in Fig. 6.1. The summarization is formalized with Algorithm 5 and serves two main purposes: firstly, it drastically reduces simulation times. Secondly, it prevents a single application with a very large request trace from dominating data bus utilisation graphs.

Algorithm 5 Summarizes a trace

```
1: // Inputs: N (number of requests of original trace), request_trace (a trace with N
   requests),
2: //              summarized_N (number of requests in the summarized trace)
3: // Output: summarized_trace[summarized_N]
4: function SUMMARIZE_TRACE(N, request_trace[N], summarized_N)
5:     summarized_trace ← new Request_Trace[summarized_N] ;
6:
7:     n_of_rm ← COUNT_NUMBER_OF_READ_MISSES(request_trace) ;
8:     n_of_wm ← COUNT_NUMBER_OF_WRITE_MISSES(request_trace);
9:     n_of_rh ← COUNT_NUMBER_OF_READ_HITS(request_trace) ;
10:    n_of_wh ← COUNT_NUMBER_OF_WRITE_HITS(request_trace) ;
11:
12:    // Computes probability of each request appearing in trace
13:    prob_rm ← n_of_rm / N;
14:    prob_wm ← n_of_wm / N;
15:    prob_rh ← n_of_rh / N;
16:    prob_wh ← n_of_wh / N;
17:
18:    for index ← 0; index < summarized_N; index ← index + 1 do
19:        new_request ← GENERATE_REQUEST(prob_rm, prob_wm, prob_rh, prob_wh)
   ;
20:        summarized_trace[index] ← new_request ;
21:    return summarized_trace ;
22:
23: function GENERATE_REQUEST(prob_rm, prob_wm, prob_rh, prob_wh)
24:    range_rm ← prob_rm ;
25:    range_wm ← prob_rm + prob_wm ;
26:    range_rh ← prob_rm + prob_wm + prob_rh ;
27:    range_wh ← prob_rm + prob_wm + prob_rh + prob_wh ;
28:
29:    aux ← GENERATE_RANDOM_FLOAT_BETWEEN_0_AND_1( );
30:
31:    if aux < range_rm then
32:        new_request ← new ReadMissRequest();
33:    else if aux < range_wm then
34:        new_request ← new WriteMissRequest();
35:    else if aux < range_rh then
36:        new_request ← new ReadHitRequest();
37:    else
38:        new_request ← new WriteHitRequest();
39:    return new_request
```



(a) EEMBC



(b) Mibench.

Figure 6.1.: Percentage of each request type for applications. The words *hits* and *misses* refer to the *row buffer* of an SDRAM bank, not to L1-cache.

Cycle-accurate simulations are now discussed. As it will become clear, a large portion of this chapter relies on the simulation of cycle-accurate models of SDRAM controllers. Consequently, the remaining of this section discusses how such simulations are performed.

A simulation is characterized by the following elements: SDRAM controller, SDRAM module, set of requestors, requestor-to-bank mapping, and halting condition. The first two are trivial: they refer respectively to which controller and to which SDRAM module are being simulated. The last three demand a more elaborate discussion and are now addressed individually.

The term requestor is used throughout this dissertation to refer to a processor running an application (also referred to as a task). In a simulation, a requestor is represented by a traffic generator whose operation is guided either by one of the *summarized traces* discussed in Section 6.1 or by a set of parameters. For the former, the traffic generator reads requests from a file containing a *summarized trace* and then sends such requests to a cycle-accurate model of a controller to which it is plugged. For the latter, the traffic generator operates according to given parameters, e.g. generates a total of 5000 requests, from which 20% are read misses, 30% are write misses, 20% are read hits and 30% are write hits.

Each requestor can be further configured as either *critical* or *best-effort*. The difference between them is as follows:

- *Critical* requestors tolerate at most one pending SDRAM request, as to mimic a processor that is timing compositional (see Section 5.1 in Chapter 5). More specifically, the traffic generator only issues a new request after its previous request is served.
- *Best-effort* requestors tolerate more than a single pending SDRAM request, as to mimic performance-oriented and non-timing-compositional processors. The exact number of tolerated pending requests depends on the experiment.

The requestor-to-bank mapping is now addressed. The term requestor-to-bank mapping refers to which bank each of the traffic generators in the simulation have access. As the applications considered in this evaluation (see Section 6.1) do not demand data sharing, *most of the simulations* rely on a *private*-bank setup, i.e. each traffic generator (including *best-effort*) gets exclusive access to one of the SDRAM banks. The expression *most of the simulations* refers to the fact that Section 6.2, which assesses the influence of intra-bank interference in cumulative SDRAM latencies, also investigates a shared bank setup.

Finally, the halting condition is discussed. The halting condition refers to which circumstances must be fulfilled for the simulation to be terminated. Those are discussed individually for each experiment. For instance, one possibility is to terminate the simulation after all traffic generators inject and receive an acknowledgment for every request in their corresponding request sets.

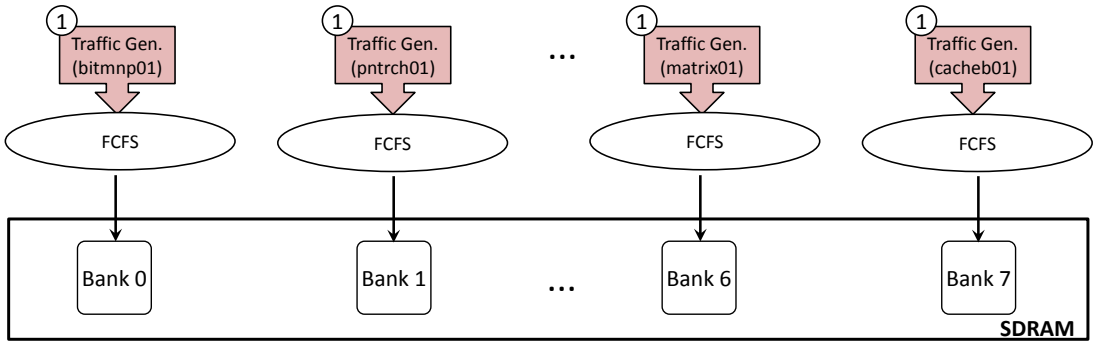
6.2. Intra-Bank Interference and Bank Privatization

The controller proposed in this dissertation (more specifically its channel scheduler) is designed so that inter-bank interference is bounded. However, in order for *critical* requestors to be isolated from the timing perspective, intra-bank interference must also be bounded. As discussed in Section 4.4, bounding intra-bank interference is achieved through a combination of requestor-to-bank mapping and, in case of bank sharing across requestors from different criticalities, monitoring of the behavior of *best-effort* requestors [1].

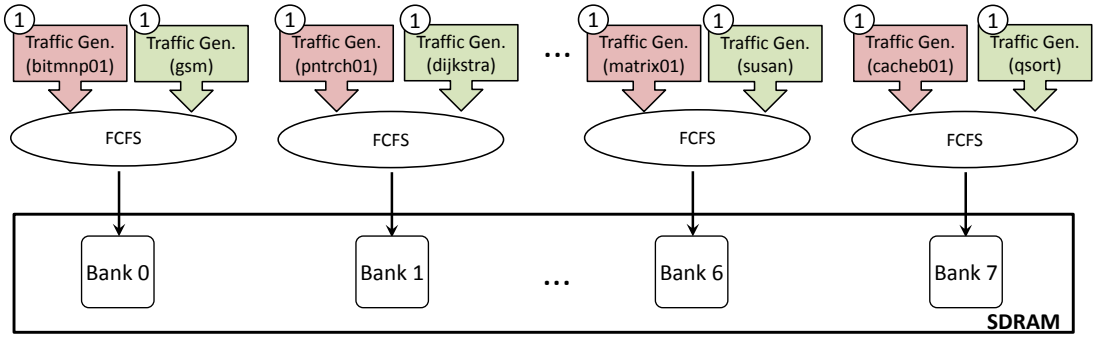
This section assesses the influence that inter-bank interference can have on the cumulative SDRAM latency of a *critical* task. For that purpose, this section considers a DDR3-1600K SDRAM module with 8 banks being operated by the controller proposed in this dissertation. As for traffic generator configuration and bank partitioning, a total of three scenarios are investigated. They are illustrated in Fig. 6.2 and enumerated below:

- Scenario 1: Only 8 *critical* requestors (that play traces from the applications in Fig. 6.1a) are considered. Each requestor has exclusive access to one of the SDRAM banks.
- Scenario 2: A total of 8 *critical* and 8 *best-effort* requestors playing traces from the applications in Figs. 6.1a and 6.1b are considered. Each SDRAM bank accom-

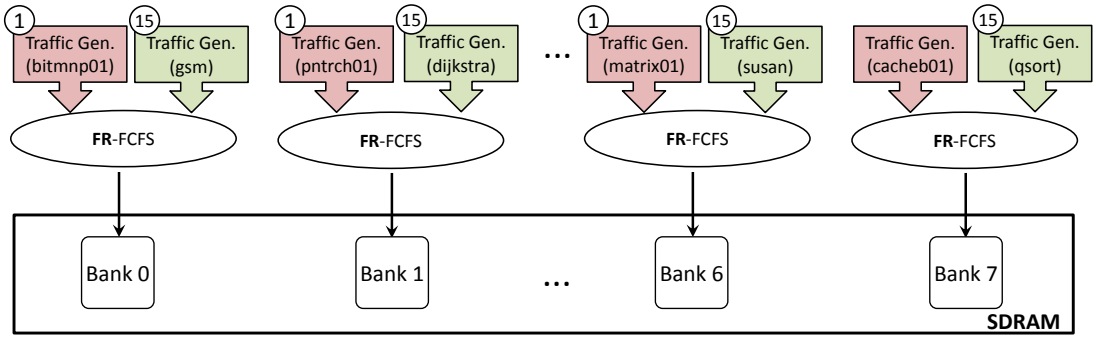
[1] If no monitoring is employed, than the controller architecture has to be modified. See Section 4.4.



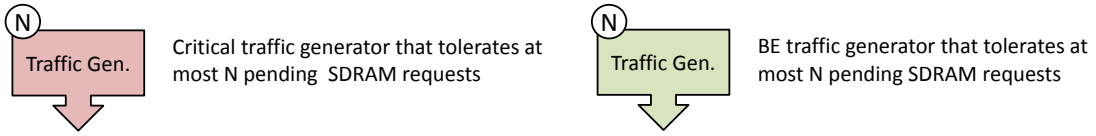
(a) Scenario 1.



(b) Scenario 2.



(c) Scenario 3.



(d) Graphical notation.

Figure 6.2.: Scenarios investigated in the experiment. Notice that the blocks labeled as FCFS and FR-FCFS refer to the policy employed inside the bank request queues of the proposed controller. The controller itself is not depicted for the sake of clarity.

modates one *critical* and one *best-effort* requestor[2]. The bank assignment follows the order depicted in Figs. 6.1a and 6.1b, i.e. *bitmnp*01 shares a bank with *gsm*, *pntrch*01 shares a bank with *dijkstra* and so on. The bank request queues in the controller are all marked as *critical* (see Section 4.1) and, as a consequence, employs FCFS. Moreover, *best-effort* traffic generators are configured to only issue a new request after the previous one has been served, a behavior that is assumed by the timing analysis of accesses to shared banks in Section 5.5.

- Scenario 3: Same requestor-to-bank assignment as Scenario 2. The bank request queues are all marked as *best-effort* (see Section 4.1) and, as a consequence, employ FR-FCFS. Moreover, the traffic generators playing traces of *best-effort* applications can tolerate up to 15 pending requests. This large number is chosen to increase the effect of request reordering made by the FR-FCFS arbiter.

In each scenario, the halting condition for the simulation is the following: all *critical* traffic generators need to inject and receive an acknowledgment for every request in their corresponding trace. During the simulations, if *best-effort* traffic generators finish before their *critical* counterparts, such *best-effort* traffic generators are restarted. The restart enforces that the interference caused by *best-effort* on *critical* never ceases until the simulation is terminated.

The three scenarios are simulated. Moreover, analytical bounds for the *critical* requestors in scenarios 1 and 2 are calculated according to the analysis presented in Chapter 5. No bounds for *critical* requestors in scenario 3 are computed because of the FR-FCFS policy and no data is collected for *best-effort* requestor because they are only being used to generate interference.

The results obtained during the simulations of the scenarios are depicted in Fig. 6.3. They are summarized with the following observations:

1. The analytical bounds for *critical* requestors in scenario 1 are always larger than the corresponding cumulative latencies measured in simulation, i.e. no violation of timing guarantees are observed.
2. The analytical bounds for *critical* requestors in scenario 2 are also always larger than the corresponding cumulative latencies measured in simulation, i.e. no violation of timing guarantees are observed. Furthermore, the analytical bounds in scenario 2 are always more than two times larger than the corresponding bounds in scenario 1. This effect is a direct result of intra-bank interference in the computation of analytical bounds. More specifically, if a *critical* requestor shares a bank with a *best-effort* requestor (that adheres to the behavior of only issuing a new request after an acknowledgment for previous requests has been received), then each *critical* request is potentially blocked once by a *best-effort* requestor. Moreover, the *row buffer* locality of the *critical* requestor, exploited to compute the analytical bounds in scenario 1, is destroyed.

[2]Notice, however, that the each pair of *critical* and *best-effort* requestor sharing a bank is not sharing data, they are simply competing for different rows of the same bank. This is because the applications used in this evaluation did not require data sharing.

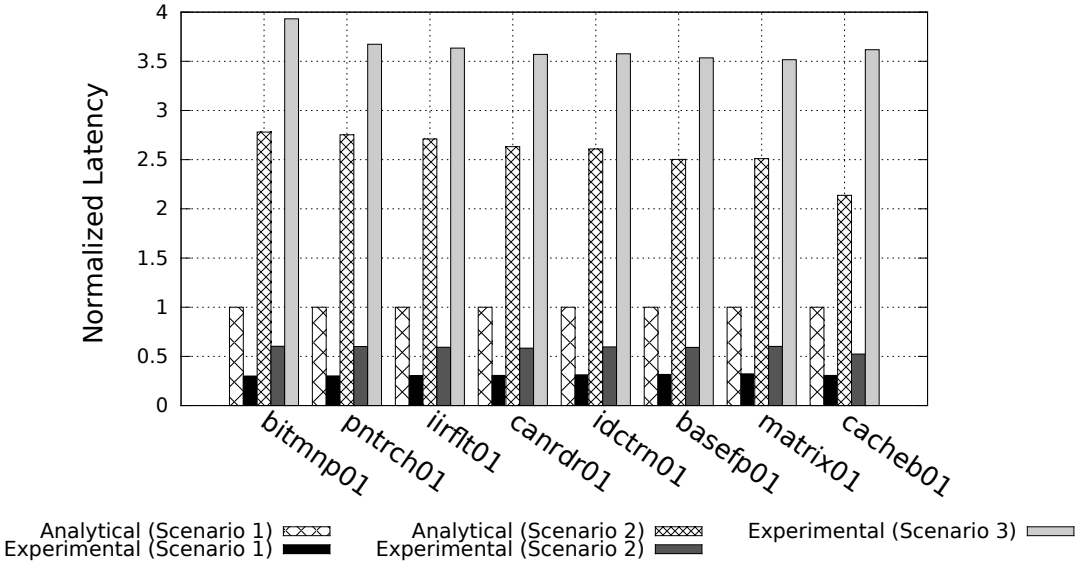


Figure 6.3.: Cumulative SDRAM latencies of *critical* applications measured in three different scenarios. For each application, results are normalized to the analytical bound computed for scenario 1.

3. The difference between analytical bounds computed in scenario 2 and the ones computed in scenario 1 are smaller for applications with a small number of *row buffer* hits, e.g. *cacheb*01. The reason for it is that, for such applications, *row buffer* locality did not exist in the first place and, hence, is not destroyed by sharing a bank.
4. The experimentally measured cumulative SDRAM latencies of *critical* requestors in scenario 2 are around two times larger than the ones in scenario 1. This is the effect of intra-bank interference.
5. The cumulative SDRAM latencies measured in scenarios 1 and 2 are significantly smaller than the corresponding analytical bounds. The reason for it is that intra-bank latencies (e.g. $dPA\text{-}RB$) and inter-bank interference (e.g. $dAA\text{-}R\overline{B}$) tend to overlap, i.e. run in parallel. Such overlap cannot be assumed by a timing analysis.
6. The cumulative SDRAM latency of *critical* requestors in scenario 3 is significantly larger than the bounds computed for scenarios 1 and 2. This is a consequence of the reordering performed by the FR-FCFS arbiter in combination of the 15 pending requests tolerated by each *best-effort* requestor. Notice that no bounds are computed for scenario 3 precisely because of the FR-FCFS policy.

From the presented results, the following conclusion is drawn: sharing a bank between different requestors should be avoided. If it is necessary, e.g. for data sharing and/or

because the number of banks in the system is smaller than the number of requestors, it should be implemented in a way that allows the extraction of timing bounds. For that purpose, the option investigated in this section, i.e. scenario 2, consisted on the following:

1. The bank request queue of the bank being shared employs the FCFS policy.
2. *Best-effort* requestors sharing a bank with a *critical* requestor are only allowed to issue a new request (to the shared bank) after their previous request is acknowledged by the controller. In a real system (i.e. a non-simulation environment) in which *best-effort* applications are executed in superscalar processors, this could be achieved with a monitoring layer (see Section 5.5).
3. Analytical bounds are computed according to what is discussed in Section 5.5.

6.3. Comparison with Related Work

This section compares the controller proposed in this dissertation with competing controllers. As the competing controllers relevant[3] for the comparison have not covered the DDR4 standard, only DDR2 and DDR3 SDRAMs are considered.

The comparison is structured as follows: firstly, the experimental setup is described. Then, results for both single- and multi-rank scenarios are discussed.

6.3.1. Experimental Setup

Given a specific SDRAM module and a set of *critical* and *best-effort* requestors, a set of simulations is performed in order to assess how the controller proposed in this dissertation and competing controllers behave both from the *critical* and *best-effort* perspectives. For each simulation, a *private*-bank setup is employed, i.e. each requestor, including *best-effort* ones, are assigned exclusive access to one of the banks in the SDRAM module. Each simulation consists in allowing the traffic generators to inject and receive an acknowledgment for every request in their corresponding trace. If every traffic generator injected its entire request trace into the controller, the simulation is over.

During each simulation, the following information is recorded:

- the observed cumulative worst-case latencies of *critical* requestors, which are compared with the corresponding analytical bounds (computed according to the analysis in Chapter 5).
- The average request latency of *best-effort* requestors.
- The data bus utilisation that each controller is able to maintain, which allows considerations about scheduling efficiency to be made.

[3] In order to focus on the benefits of read/write bundling, the controllers selected for the comparison have the following similarities: firstly, *critical* requestors have exclusive access to one of the SDRAM banks. And secondly, the distinction between *critical* and *best-effort* is made at the request-level and not at the command-level. Such features exclude, for instance, Predator, which relies on an *interleaved* request-to-bank mapping. Similarly, they also exclude DCmc and *CMD-priority*, which prioritize commands from *critical* requests over commands from *best-effort* requests.

- SDRAM command traces, which serve as input for a power estimation tool (DRAMPower [18]). As the DRAMPower tool did not support multi-rank modules at the time of writing of this dissertation, the power comparison is made exclusively for single-rank systems.

The results for single- and multi-rank systems are presented in Sections 6.3.2 and 6.3.3, respectively.

6.3.2. Single-Rank Systems

The set of requestors employed for the evaluation of single-rank controllers is as follows:

- Four *critical* requestors, represented by the following applications from EEMBC: *bitmnp*01, *pntrch*01, *matrix*01 and *cacheb*01.
- Four *best-effort* requestors, represented by the following applications from Mibench: $tiff2bw$, $tiff2rgba$, *susan* and *qsort*.

As for modules, a total of 5 different are considered and they are built respectively out of DDR2-800E, DDR3-1333H, DDR3-1600K, DDR3-1866M and DDR3-2133N devices. All have a 64-bit data buses and a total of 8 banks (notice that DDR2 and DDR3 have no *bank groups*). As for controllers, in addition to the one proposed in this dissertation, two other are investigated:

- the ORP [126]. As discussed in Chapter 3, the ORP uses the *open-row* policy and assumes that the task under consideration has exclusive access to one of the banks, i.e. it considers bank privatization. Its main difference in comparison with the controller proposed in this dissertation is that older CAS commands always have priority over newer ones, regardless of whether they force a bus turnaround or not. Although the article that introduced ORP did not mention mixed criticality, banks assigned to *best-effort* requestors can employ FR-FCFS policy to maximize *row buffer* exploitation without compromising guarantees for *critical* requestors. Such modification is employed for the sake of this evaluation.
- The *Analyzable Memory Controller (AMC)* [97, 96]. The AMC employs the *close-row* policy and originally relied on a *interleaved* request-to-bank mapping. However, with a 64-bit data bus (the scenario considered in this evaluation), an *interleaved* request-to-bank mapping is not useful when SDRAM requests have the size of a cache line. Hence, in order to perform a comparison, this evaluation employs the strategy proposed in [126]: AMC is implemented with a *private* bank setup in which each incoming request ultimately is translated into a static command group containing an *activate*-(CAS with Auto-Precharge) sequence. The oldest pending command group, regardless of whether it forces a bus turnaround or not, is given priority. No distinction is made between *best-effort* and *critical*.

For each combination of controller and SDRAM module (the set of requestors is fixed), a simulation is performed and the data described in Section 6.3.1 is collected. The obtained results are now individually presented and discussed.

6.3.2.1. Worst-case Cumulative SDRAM Latency (Critical)

The cumulative worst-case latencies of the *critical* requestors (both analytical and experimental) are depicted in Figs. 6.4a, 6.4b, 6.4c, 6.4d and 6.4e. For each application in each of the figures, all results are normalized to the analytical bound obtained for AMC.

The following trends are observed:

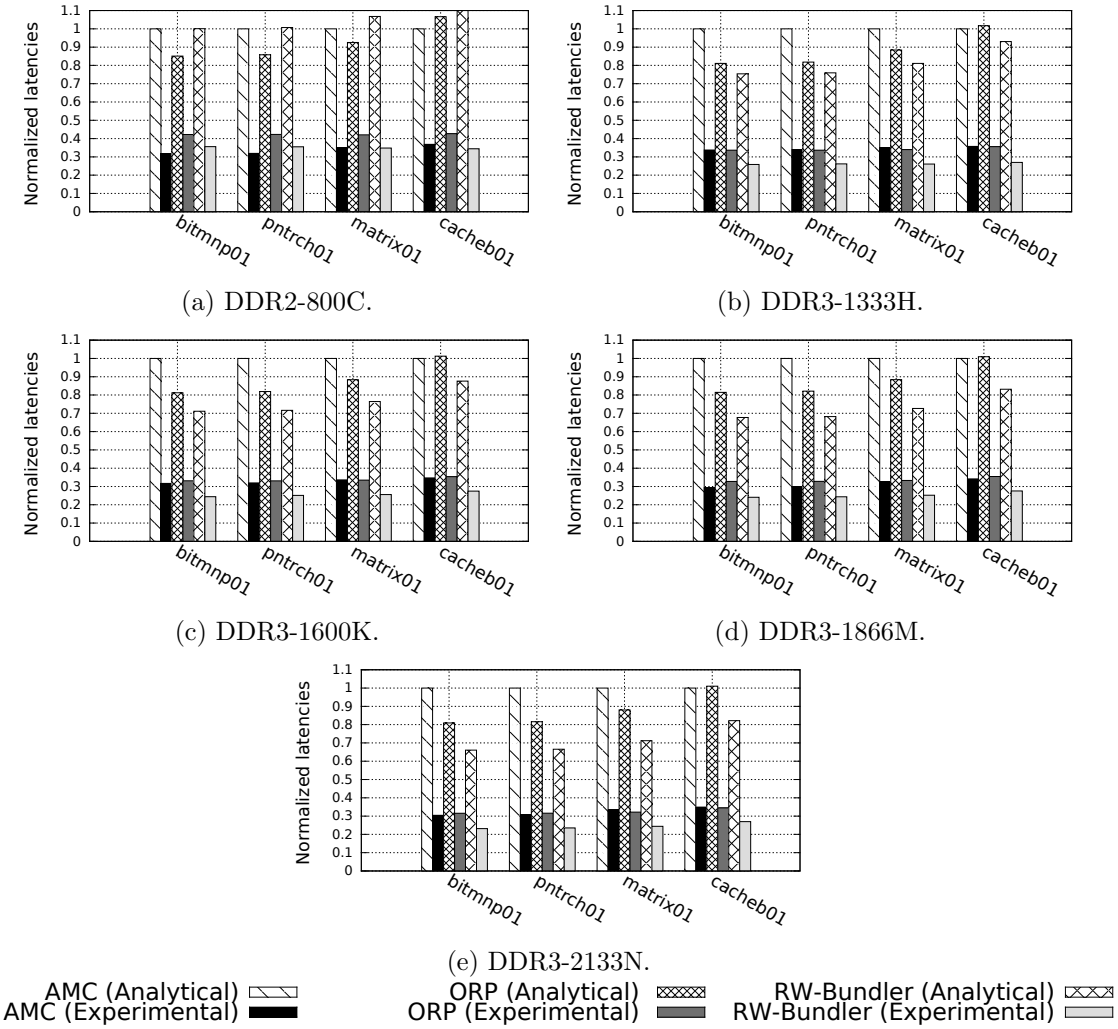
1. Regardless of the controllers, the cumulative SDRAM latencies measured in simulation are smaller than the computed analytical bounds.
2. For the *open-row* controllers, applications that have a larger number of *row buffer* hits, such as *bitmnp01*, have smaller timing bounds than the ones with a low number of *row buffer* hits, such as *cacheb01*. This is because requests that hit in the *row buffer* do not demand *activate* and *precharge* commands, which improves their latencies.
3. The analytic advantage of *open-row* controllers (ORP and the RW-Bundler) over the *close-row* one (AMC) is larger in high-speed modules. This is because in high-speed modules, the number of cycles for closing and opening rows is larger, a phenomenon discussed in Section 2.1.2.
4. Except for the DDR2-800C module, the RW-Bundler provides better timing bounds than the other controllers. This is largely because the analysis of other controllers must assume an alternating pattern of interfering read and write requests to compute guarantees. In the RW-Bundler, such scheduling pattern is architecturally prevented.
5. The advantage of the RW-Bundler is better highlighted in high-speed modules. This is because the penalty for data bus turnarounds is larger in high-speed modules.

6.3.2.2. Average Request Latency (Best-effort)

The average request latencies of the *best-effort* requestors are depicted in Figs. 6.5a, 6.5b, 6.5c, 6.5d and 6.5e. For each application in each figure, results are normalized to the value obtained for AMC.

The following trends are observed:

1. As for *open-row* controllers the *best-effort* banks are configured to employ the FR-FCFS (which increases the number of *row buffer* hits of a requestor), the *open-row* controllers perform better than the AMC for all modules with the exception of DDR2-800C.
2. The difference in performance between the *open-row* controllers and the AMC is larger for faster SDRAM modules. This is because the number of cycles to open and close a *row buffer* is larger for faster devices, a phenomenon discussed in Section 2.1. As a matter of fact, for DDR2-800C, for which the number of cycles to open and close *row buffers* is small, the difference between the *open-row* and the *close-row* controllers is barely noticeable.



(a) DDR2-800C.

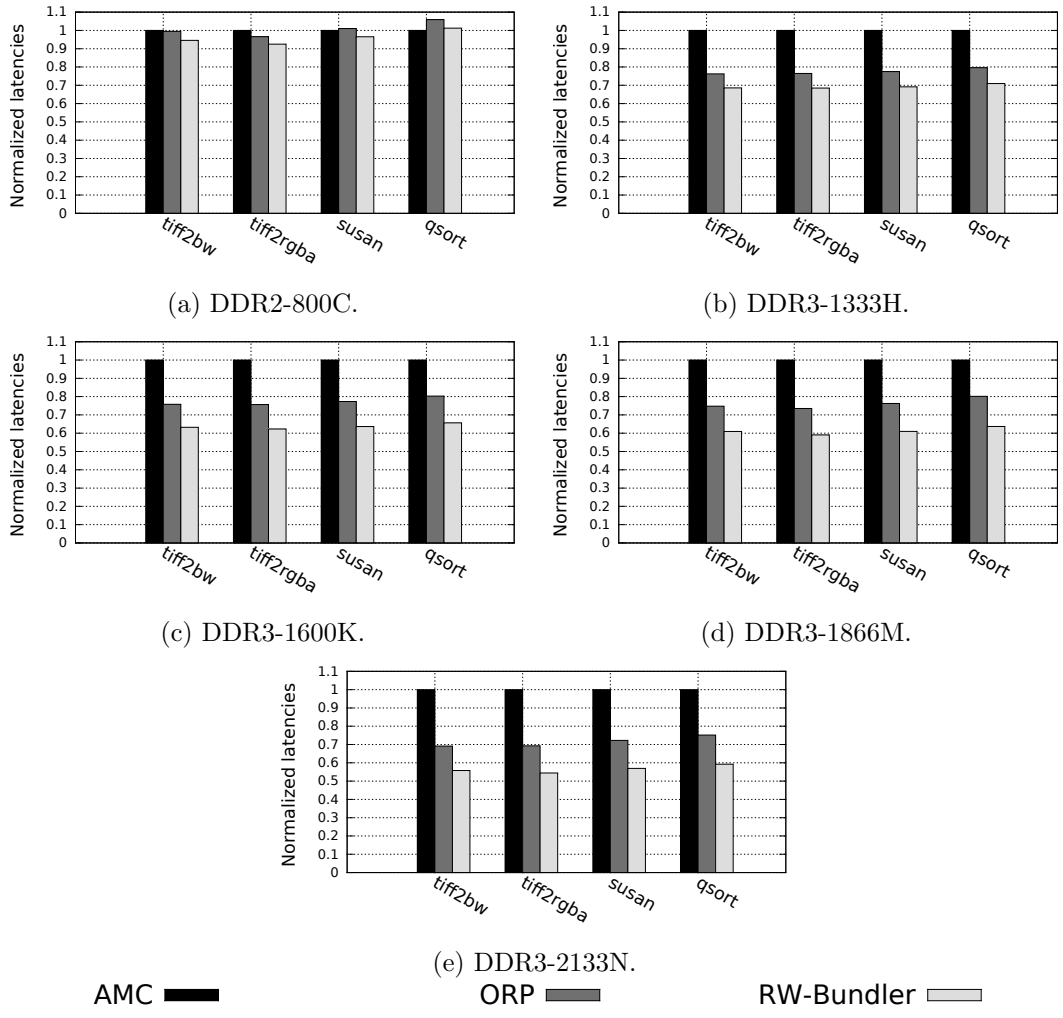
(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.4.: Comparison of cumulative SDRAM latencies for *critical* requestors in single-rank controllers ($nB = 8$, $nG = 1$ and $nR = 1$).



(a) DDR2-800C.

(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.5.: Comparison of average request latency for *best-effort* requestors single-rank controllers ($nB = 8$, $nG = 1$ and $nR = 1$).

3. Considering only the *open-row* controllers, the RW-Bundler improves the average request latency in comparison with the ORP because it minimizes the number of data bus turnarounds.

6.3.2.3. Data Bus Utilisation

The data bus utilisation that each controller is able to maintain for each of the five SDRAM modules is depicted in Figs. 6.6a, 6.6b, 6.6c, 6.6d and 6.6e. Notice that the figures measure time in data bus clock cycles (and not in nanoseconds). Hence, even though using a DDR2-800C takes less data bus cycles than a DDR3-1866M to serve the same set of requests, the latter takes less nanoseconds, as it has a clock period of only 1.07 ns, while the former has a period of 2.5 ns.

The following trends are observed:

1. regardless of the controller, it becomes harder to keep a high data bus utilisation for faster SDRAM modules, e.g. compare Fig. 6.6a and Fig. 6.6e. For instance, the utilisation maintained by the RW-Bundler in the DDR2-800C module is close to 90%. Such value drops to around 65% in the DDR3-2133N module. As discussed in Section 2.1, this is because the timing constraints are larger for faster devices.
2. For DDR2-800C, the AMC actually performs better than ORP. As a matter of fact, its utilisation mostly overlaps with the one displayed by the RW-Bundler. This is because the overhead to close and open rows is smaller for devices with reduced operating frequency.
3. For the remaining modules, AMC performs worse than the ORP. Moreover, regardless of the SDRAM module, the RW-Bundler consistently maintains higher utilisation than the other two investigated controllers and, as a consequence, finishes serving the workload earlier. (Notice that the drop to 0% in data bus utilisation marks the end of the simulation.)

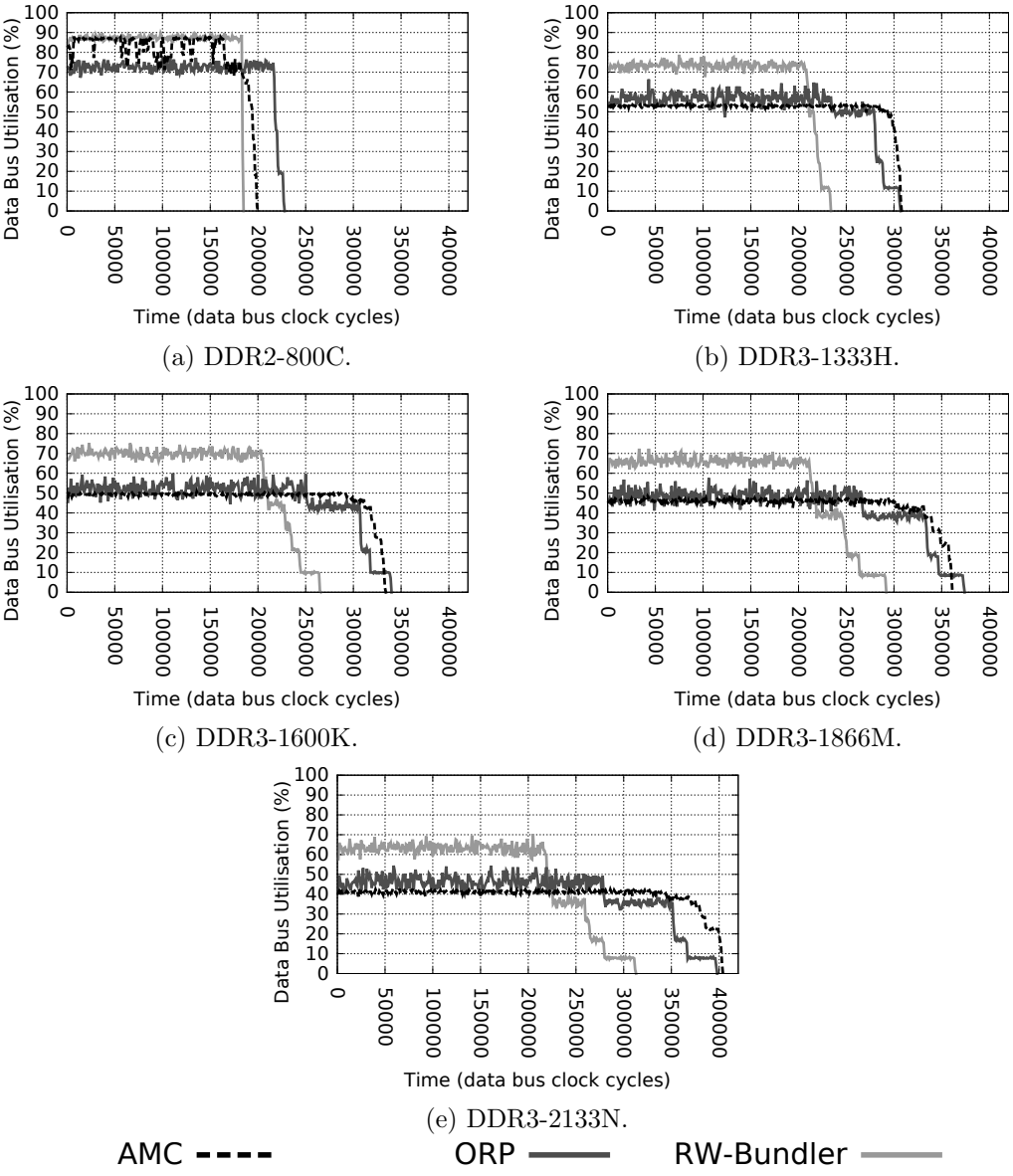
6.3.2.4. Power Consumption

In order to perform a power comparison, the DRAMPower tool [18] demands a command trace and a file describing the electrical parameters of a SDRAM module. The former is obtained using the cycle-accurate simulations described in Section 6.3.1. The latter can be obtained from a SDRAM module manufacturer, such as Micron [104].

So that the correct electrical parameters can be retrieved, the part number and die revision of the modules is described in Table 6.1. Notice, however, that no DDR3-2133N is in the table. This is because at the time of writing of this dissertation, Micron did not provide DDR3-2133N modules.

Finally, using the same command traces employed in the last subsection, a power consumption estimate is computed using the DRAMPower tool [18]. The results are depicted in Fig. 6.7 and represent the total amount of energy (in micro joules) required to serve all requests.

The following trends are observed:



(a) DDR2-800C.

(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.6.: Comparison of data bus utilisation for single-rank controllers ($nB = 8$, $nG = 1$ and $nR = 1$). The drop to 0% utilisation marks the moment in which all requests from the workload are served.

Table 6.1.: Specification of SDRAM Modules from Micron [104].

DDR Gen.	Model	Capacity	Part Number	Die Rev.	Vdd
DDR2	800C	1 GB	MT8HTF12864A(I)Z-80E	G	1.80 V
DDR3	1333H	1 GB	MT8KTF25664AZ-1G4	K	1.35 V
DDR3	1600K	1 GB	MT8KTF12864AZ-1G6	J	1.35 V
DDR3	1866M	2 GB	MT4KTF25664AZ-1G9	P	1.35 V

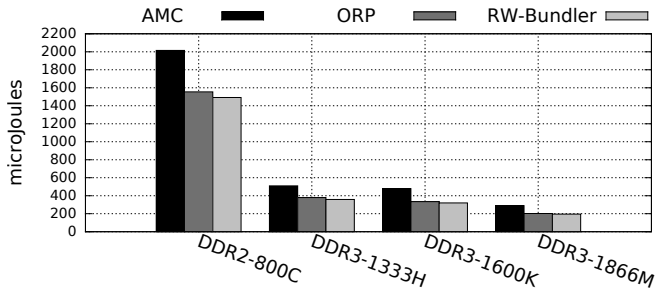


Figure 6.7.: Power consumption.

1. regardless of the controller, DDR2-800C has by far the worst power consumption. This is because DDR2 memories are simply not as energy efficient as DDR3. Moreover, they have an operating voltage of 1.8 V (see Table 6.1), against 1.35 V of the DDR3 investigated modules.
2. For all SDRAM modules, the AMC consumes more power than the *open-row* controllers. This is because closing and opening rows is an energy-costly operation.
3. For all SDRAM modules, the RW-Bundler provides a small power consumption reduction over ORP. This is because it serves the requests of a workload faster (see Fig. 6.6). Hence, the amount of static power dissipated is smaller.

6.3.3. Multi-Rank Systems

The set of requestors employed for the evaluation of multi-rank controllers is as follows:

- eight *critical* requestors, represented by the EEMBC applications listed in Fig 6.1a.
- eight *best-effort* requestors, represented by the Mibench applications listed in Fig. 6.1b.

As for modules, a total of 5 different are considered and they are built respectively out of DDR2-800E, DDR3-1333H, DDR3-1600K, DDR3-1866M and DDR3-2133N devices. All have a 64-bit data buses and a total of 16 banks divided into two ranks . As for controllers, in addition to the one proposed in this dissertation, ROC is considered (see Chapter 3). ROC is a non-pattern-based controller that also uses the *open-row* policy. With regard to the controller proposed in this dissertation, ROC employs round-robin

to alternate the use of the data bus between ranks.

For each combination of controller and SDRAM module (the set of requestors is fixed), a simulation is performed and the data described in Section 6.3.1 is collected. The obtained results are now individually presented and discussed. Be aware, however, that no power comparison is presented because at the time of writing, the DRAMPower tool did not support multi-rank modules. Moreover, notice that no *close-row* controller is employed in this comparison[4].

6.3.3.1. Worst-case Cumulative SDRAM Latency (Critical)

The cumulative worst-case latencies of the *critical* requestors (both analytical and experimental) are depicted in Figs. 6.8a, 6.8b, 6.8c, 6.8d and 6.8e. For each application in each of the figures, all results are normalized to the analytical bound obtained for ROC.

The following trends are observed:

1. For slow SDRAM modules, i.e. the DDR2-800C and the DDR3-1333H, ROC provides better timing bounds than the RW-Bundler. This is because the penalty for rank switches is smaller for such modules (see Section 2.2).
2. For the two SDRAM modules mentioned in the previous item, the cumulative SDRAM latencies measured for the RW-Bundler are smaller than the ones from ROC (although analytically ROC provides smaller timing bounds). This indicates that the analysis for the RW-Bundler in multi-rank setups lacks tightness and is largely a consequence of the behavior of *too-late* commands in such scenarios. More specifically, *too-late* CAS commands can be blocked twice by each command register in interfering ranks (see Lemma 5.6 in Section 5.2.1.3).
3. For the remaining modules, the RW-Bundler provides better analytical and experimental cumulative SDRAM latencies than ROC. Moreover, the improvement of the RW-Bundler over ROC is larger for the DDR3-2133N module. This is because, for such module, the rank switching penalties and the data bus turnarounds[5] are larger (see Chapter 2).

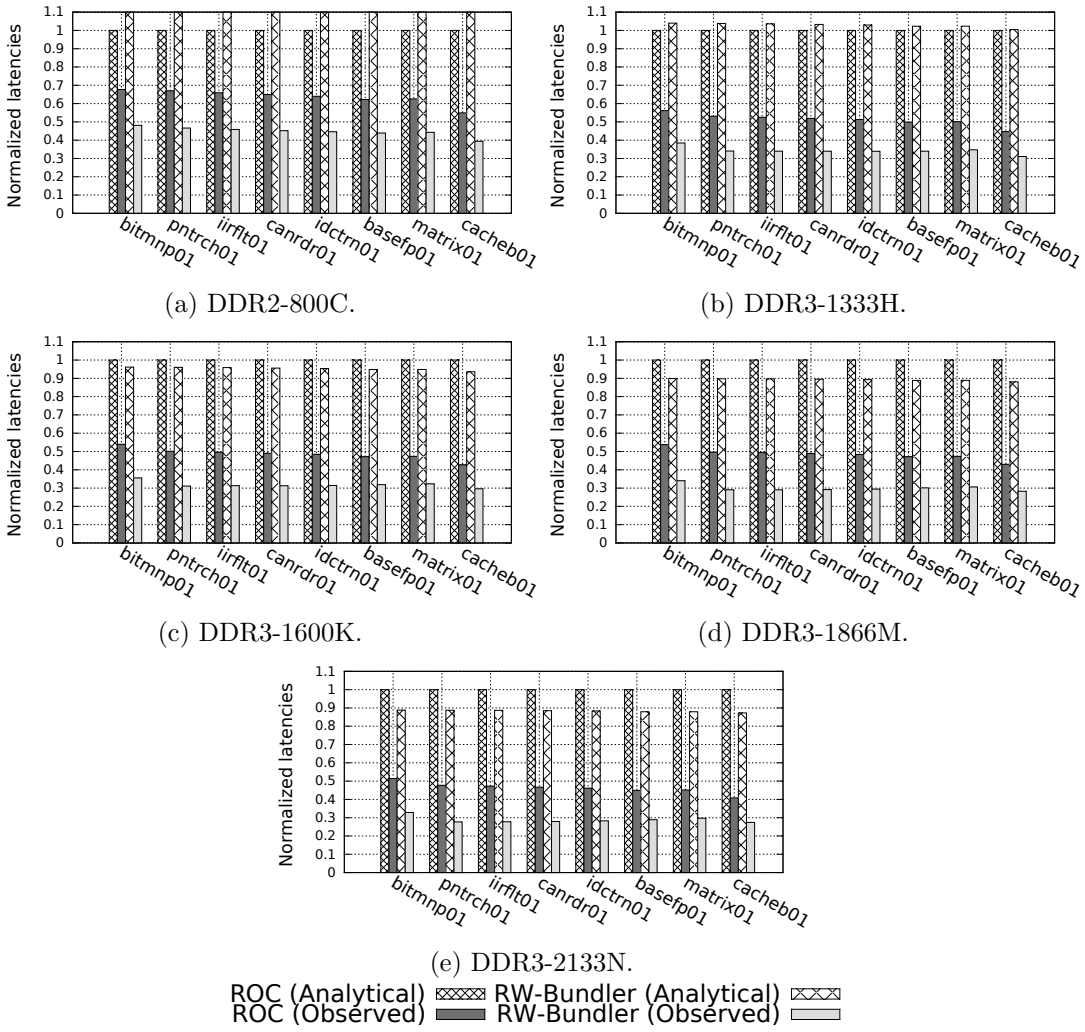
6.3.3.2. Average Request Latency (Best-effort)

The average request latencies of the *best-effort* requestors are depicted in Figs. 6.9a, 6.9b, 6.9c, 6.9d and 6.9e. For each application in each of the figures, all results are normalized to the latency obtained for ROC.

The following trends are observed:

[4]PRET could represent the *close-row* controller-category in the comparison. However, the article that introduced PRET computed its command patterns in a speed-bin specific fashion. Due to the pressing deadline to finish this dissertation, the author deemed the effort to design an algorithm to generate such patterns independently of speed-bin unfeasible.

[5]Notice that although a data bus turnaround is an intra-rank event, its latency may not be completely hidden by a rank switch. This means that ROC, which alternates the control of the data bus between ranks using round-robin, can perform scheduling that reduces data bus utilisation not only because of rank switches, but also because of turnarounds.



(a) DDR2-800C.

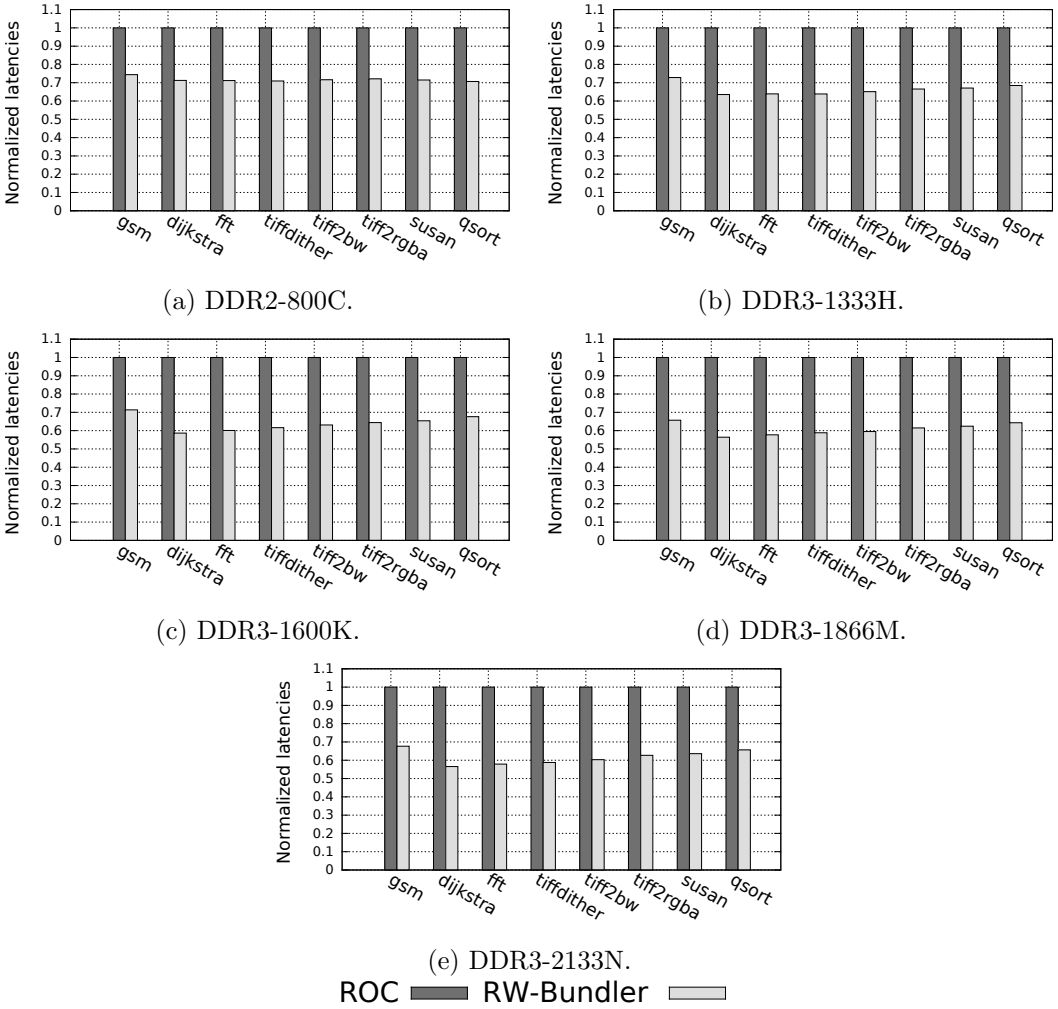
(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.8.: Comparison of cumulative SDRAM latencies for *critical* requestors in multi-rank controllers ($nB = 8$, $nG = 1$ and $nR = 2$).



(a) DDR2-800C.

(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.9.: Comparison of average request latency for *best-effort* requestors in multi-rank controllers ($nB = 8$, $nG = 1$ and $nR = 2$).

1. Because it minimizes ranks switches (and data bus turnarounds[6]), the RW-Bundler improves the average latency of *best-effort* requestors.
2. The difference in performance between the two controllers is larger for faster SDRAM modules. This is because the overhead for rank switches (and data bus turnarounds[7]) is larger for such modules.

6.3.3.3. Data Bus Utilisation

The data bus utilisation that each controller is able to maintain for each of the five SDRAM modules is depicted in Figs. 6.10a, 6.10b, 6.10c, 6.10d and 6.10e. As it was the case for the single-rank comparison, notice that the figures measure time in data bus clock cycles (and not in nanoseconds).

The following trends are observed:

1. As it was the case for the single-rank setup, it becomes harder to keep high data bus utilisation for faster SDRAM modules, e.g. compare Fig. 6.10a and Fig. 6.10e. This is not only because the intra-rank timing constraints are larger, but also because the rank switching overhead is larger (see Section 2.2).
2. The RW-Bundler consistently maintains higher utilization than ROC because it minimizes the number of rank switching events.

6.4. Performance Trends Across Different DDR Devices and Generations

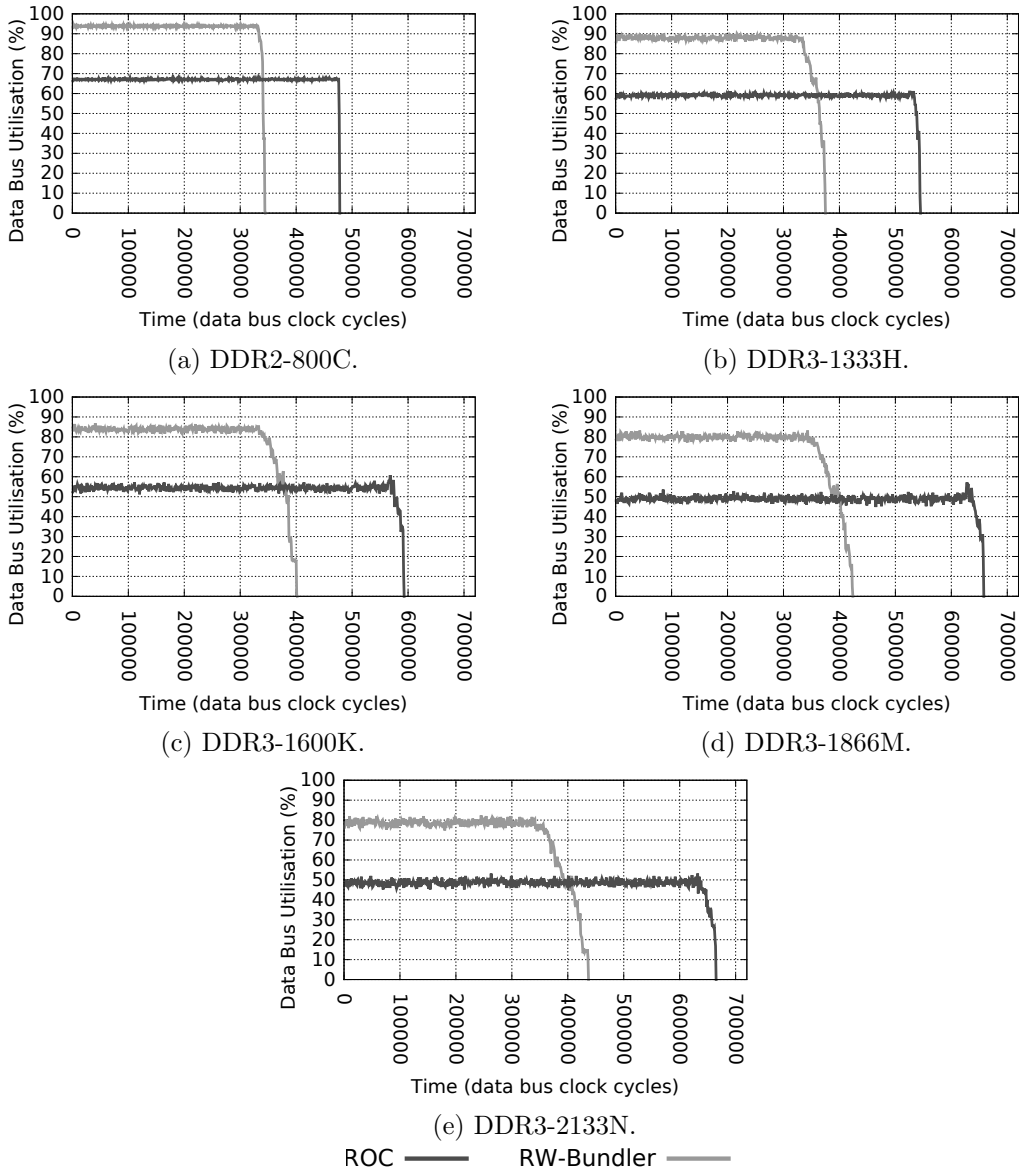
This section presents an assessment of the performance trends across different DDR SDRAM devices and speed bins. As the controllers investigated in the previous section (comparison with the related work) do not cover the DDR4 standard, this section focuses exclusively on the controller proposed in this dissertation. The remaining of this section firstly discusses the experimental setup and then discusses the obtained results.

6.4.1. Experimental Setup

The goal of this portion of the evaluation is to assess how the worst-case behavior of a *critical* requestor changes throughout a set of SDRAM modules built out of devices from different generations and speed bins, assuming the same SDRAM controller is employed. For that purpose, given a single requestor (i.e. a request trace), the the L_{Task}^{SDRAM} is computed for a wide variety of SDRAM modules built using DDR2, DDR3, and DDR4 devices. Then, cycle-accurate simulations are performed and the observed cumulative latencies are compared with the analytical bounds.

For the simulations, the requestor under analysis is always given exclusive access to one of the banks. The other banks are occupied by interference generators that trigger back-

[6] See Footnote 5.
[7] See Footnote 5.



(a) DDR2-800C.

(b) DDR3-1333H.

(c) DDR3-1600K.

(d) DDR3-1866M.

(e) DDR3-2133N.

Figure 6.10.: Comparison of data bus utilisation for multi-rank controllers ($nB = 8$, $nG = 1$ and $nR = 2$). The drop to 0% utilisation marks the moment in which all requests from the workload are served.

to-back requests[8], i.e. they issue a new request as soon as the previous one is served. The generators are programmed so that each request has a 40%, 40%, 10% and 10% probability of being a read hit, write hit, read miss and write miss, respectively. Such settings are chosen because the complexity of the controller proposed in this dissertation mainly regards the scheduling of CAS commands (and not *activates* and *precharges*). Moreover, the high ratio of writes is meant to cause frequent turnarounds (notice that, in this section, the traces from Mibench applications in Fig. 6.1b are not employed as interference because they have a smaller ratio of writes).

Finally, the modules investigated in this portion of the evaluation are identified with a string with the DDR generation, model and a suffix that describes its structure. For instance, DDR4-2400U-2r,2g,8b refers to a dual-rank module built using DDR4-2400U devices with 8 banks divided into 2 *bank groups* ($nR = 2$, $nG = 2$,$nB = 8$).

6.4.2. Results

Two applications are considered for the evaluation: *gsm*, from Mibench, which contains a high number of *row buffer* hits, and *cacheb01*, from EEMBC, which contains a low number of *row buffer* hits (see Fig. 6.1). Notice that in the previous portion of the evaluation, more specifically in the comparison with the related work (Section 6.3), applications from Mibench were used to represent *best-effort* requestors.

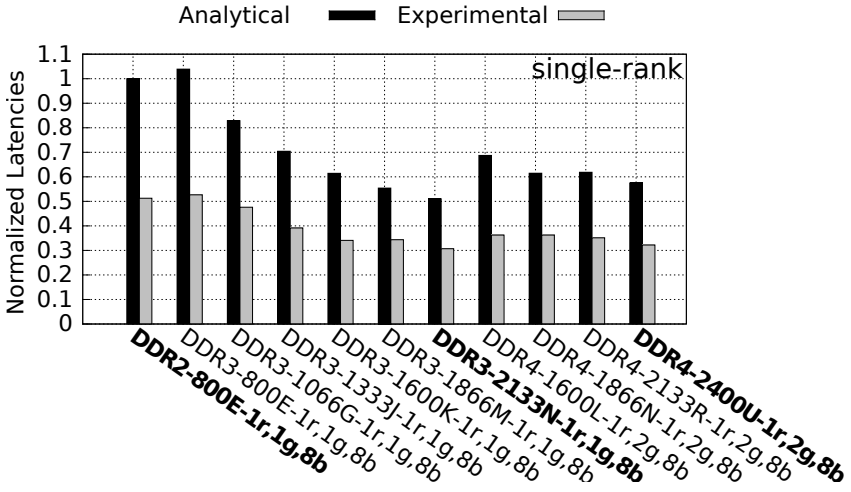
In this portion of the evaluation, however, *gsm* is used to represent a *critical* requestor. The reason for it is that from the perspective of observing trends, it is useful to compare an application with a large number of *row buffer* hits with an application with a small number of hits. With that regard, around 80% of the requests from *gsm* hit in the *row buffer* (significantly more than any application from EEMBC). Moreover, *cacheb01* has the lowest number of hits in the *row buffer* from all applications in Fig. 6.1.

Analytical bounds for *gsm* and *cacheb01* are computed and experimental simulations according to what is described in Section 6.4.1 are performed. Firstly, considering systems with a total of 8 banks ($nB \cdot nR = 8$) and then for systems with 16 banks ($nB \cdot nR = 16$). The results for systems with 8 and 16 banks are depicted in Fig. 6.11 and 6.12, respectively.

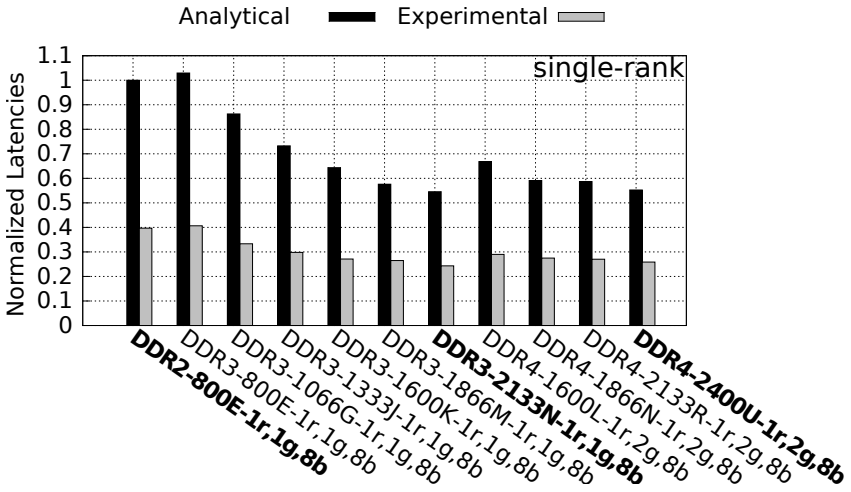
The following observations about the observed trends (for both 8- and 16-bank systems) are made:

1. The cumulative SDRAM latencies observed during the experimental simulation are always smaller than the analytical bounds.
2. Because CAS commands cannot always be executed in a *group interleaved* pattern in DDR4 systems (see Lemma 5.1 in Section 5.2.1.1), DDR4 SDRAMs perform worse than DDR3 SDRAMs both from the perspective of worst-case bounds and from the perspective of experimental simulation results. Such statement remains true even when comparing devices with different operating frequencies, e.g. DDR3-2133N and DDR4-2400U.

[8] Each interference generator issues a new request as soon as the controller acknowledges the completion of the previous request.

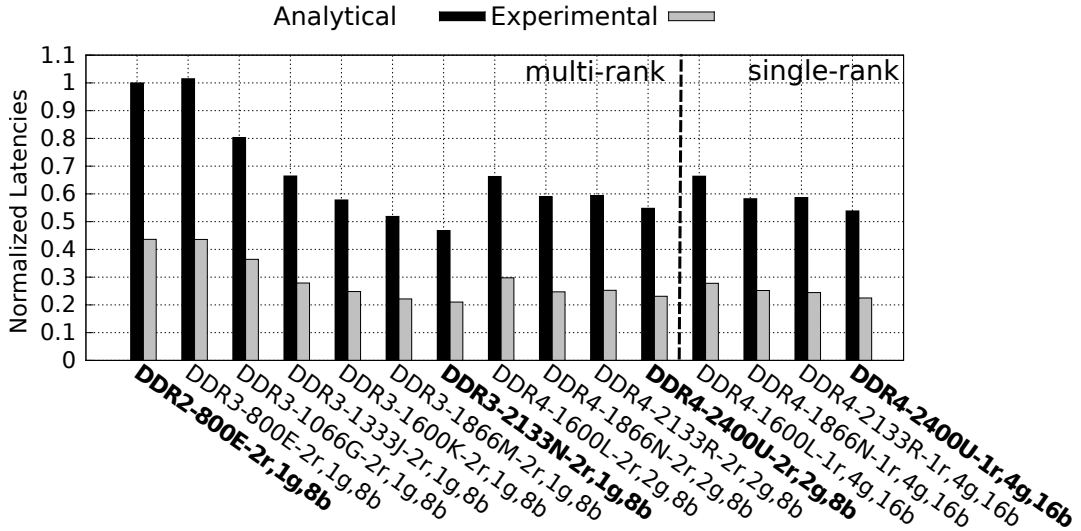


(a) *Gsm.*

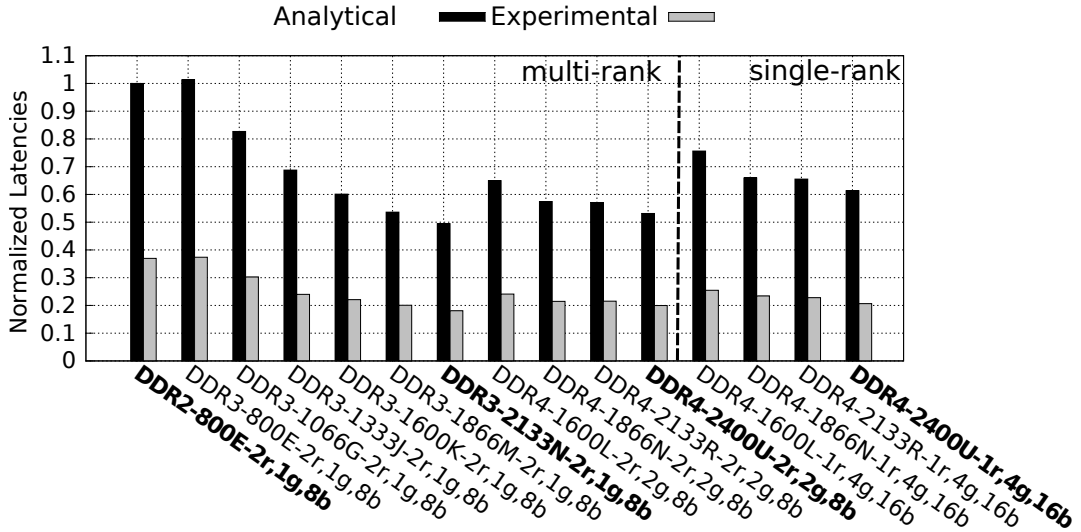


(b) *Cacheb01.*

Figure 6.11.: Worst-case cumulative latency of *gsm* and *cacheb01* applications over different modules with 8 banks. In (a) and in (b), results are normalized to the one obtained for the leftmost module (DDR2-800E-1r,1g,8b). Moreover, the smallest analytical and experimental latencies obtained for DDR2, DDR3 and DDR4 are **highlighted**.



(a) *Gsm.*



(b) *Cacheb01.*

Figure 6.12.: Worst-case cumulative latency of *gsm* and *cacheb01* applications over different modules with 16 banks. In (a) and (b), all results are normalized to the one obtained for the leftmost module, i.e. DDR2-800E-2r,1g,8b. Moreover, the smallest analytical and experimental latencies obtained for DDR2, DDR3 and DDR4 are **highlighted**.

3. And finally, the worst-case bounds for *gsm*, which displays high *row buffer* hit ratio, are tighter (closer to the results obtained experimentally) than the ones for *cacheb01*, which displays low *row buffer* hit ratio. This is partially because only 20% of the requests made by interference generators demand *precharges* and *activates*. But mainly because during the experimental simulation, intra-bank latencies tend to overlap with inter-bank interference (a behavior that cannot be assumed by the timing analysis). For the interested reader, [43] discusses architectural support to enforce that such overlap occurs.

Now the results for 16 bank systems are addressed exclusively. Such systems can only be implemented as a dual-rank setup if DDR2 or DDR3 are used, but can be implemented either as a single- or a dual-rank setup if DDR4 is used. With that respect, two further observations are made:

1. For *gsm*, the difference between the worst-case bounds obtained with single- and dual-rank DDR4 is small. This is because the largest source of inter-bank interference regards the data bus, as most requests only require CAS commands due to the large *row buffer* hit ratio of the application (see Fig. 6.1).
2. For *cacheb01*, however, the worst-case bounds are noticeably worse for the single-rank DDR4. This is because the application displays a low *row buffer* locality and, hence, several of its requests demand *activate* commands, which in turn have better worst-case latency in the dual-rank setup, as less occurrences of t_{FAW} must be accounted for (see Theorem 5.10). Nevertheless, from the perspective of experimental results, no large difference between the single- and multi-rank setups were observed (again because intra-bank latencies tend to overlap with inter-bank interference).

6.5. Summary

This chapter evaluates the controller proposed in Chapter 4 and analyzed in Chapter 5. The evaluation is based on SDRAM request traces derived from Mibench and EEMBC applications. It is comprised of three main portions: (1) an evaluation of intra-bank interference, (2) a comparison with other controllers available in the literature and (3), an evaluation of worst-case performance trends between SDRAM modules from different generations and speed bins.

In summary, the results presented in this chapter can be summarized with the following observations:

- Bank sharing leads to intra-bank interference, which worsens cumulative latency of *critical* requestors both from the analytical and experimental perspectives. Intra-bank interference can, however, be upper bounded if the assumptions from Section 5.5 are held.
- As the operating frequency of SDRAM devices and modules increases, so do the penalties to perform data bus turnarounds and rank switches. With that regard, the reordering of CAS commands proposed in this dissertation can improve worst-case performance while simultaneously improving average performance. Hence,

the proposed controller fulfills the needs of both the *best-effort* and of the *critical* requestors described in the introduction (Section 1.1.1 from Chapter 1).

- The extra complexity introduced by the *bank groups* feature in DDR4 SDRAM module complicates command scheduling, which as a consequence compromises worst-case and observed performance with regard to DDR3 SDRAM modules from the same operating frequency[9].
- Using the *open-row* policy in combination with a *private bank* mapping reduces the number of *activate* and *precharge* commands a SDRAM controller executes. This not only improves worst-case performance, but also leads to a reduction in power consumption.

[9] Here it is to be highlighted that the third portion of the evaluation focused on *critical* requestors and did not investigate trends for *best-effort* ones. However, *best-effort* requestors also experience a similar effect in scenarios with DDR4 SDRAM modules and the controller proposed in this dissertation. The reason for it is that the *bank groups* feature plays a role in inter-bank scheduling (which is implemented in the channel scheduler) and not intra-bank scheduling (where the distinction between *critical* and *best-effort* is made).

7. Concluding Remarks

This dissertation proposes a multi-generation SDRAM controller architecture for mixed criticality systems. The controller is multi-generation because command scheduling and timing analysis are presented in terms of minimum distances between consecutive commands, which abstract architectural and timing details of SDRAM devices from different generations and speed bins. At the same time, the controller is suitable for mixed criticality because although it performs scheduling optimizations to increase data bus utilisation and improve the average latency of *best-effort* requests, such optimizations do not compromise real-time guarantees for *critical* requestors (provided that intra-bank interference is controlled as described in Section 4.4).

In order for design choices made in this dissertation to be justified, Chapter 2 firstly provides a detailed overview of SDRAM chips and modules. Moreover, it pinpoints two challenges of growing relevance for the design of SDRAM controllers for the mixed criticality domain:

1. The first challenge is the data bus turnaround time. In SDRAMs, a single data bus is used for read and write operations. Hence, alternating the execution of read and write commands is highly undesirable, as it takes time to reverse the direction of the data bus, i.e. the data bus turnaround time. In COTS systems, which are purely performance oriented, the challenge is tackled by buffering write commands and executing them in a batch once the number of write commands reaches a predetermined number. In mixed criticality systems, however, the same strategy is not acceptable, as it leads to hard-to-predict behaviors.
2. The second challenge is the rank-to-rank switching time and only affects multi-rank modules. A rank, in SDRAM jargon, refers to a set of chips operating under the same clock, chip-select and command bus. A multi-rank module refers to a printed-circuit board containing two or more ranks, which in turn share the same multi-drop data bus. The rank-switching time refers to the time interval required to alternate the control of the data bus between different ranks (i.e. drivers), which is usually observed in any multi-drop bus. As a consequence, blindly alternating the execution of CAS commands between different ranks decreases data bus utilisation.

Chapter 3 then shows that controllers proposed in the literature do not efficiently[1] tackle the aforementioned challenges.

Based on the discussion of the related work, Chapter 4 proposes a SDRAM controller

[1] Here the author of this dissertation highlights that pattern-based controllers such as Predator [5] can mitigate the impact of data bus turnarounds if the ratio between request granularity and data bus width is very large, a scenario not investigated in this dissertation.

architecture that minimizes data bus turnarounds and rank switching events. In the controller, mixed criticality is implemented at the request-level. In banks assigned as *best-effort*, which are occupied exclusively by *best-effort* requestors, aggressive intra-bank request reordering is implemented in order to exploit the spatial locality of the corresponding *row buffers*. In banks assigned as *critical*, which are occupied either by *critical* or by a combination of *critical* and *best-effort* requestors, no reordering is implemented. After a request is translated into a set of commands, however, such commands compete equally with other pending commands in the system (no distinction is made between *critical* and *best-effort*).

From the perspective of command scheduling, pending commands are arbitrated in two layers: firstly, within their own type arbiters, i.e. CAS commands compete with other CAS commands in the CAS Arbiter, while *activates* and *precharges* compete with other *activates* and *precharges* in the AP Arbiter. Then, the commands that win the arbitration in their corresponding type arbiters compete with each other in the Command Bus Arbiter. This compartmentalized approach allows scheduling optimizations for a type of command to be performed without making considerations about other command types. For instance, the read/write bundling, which minimizes the number of turnarounds and rank switches, is implemented in the CAS Arbiter.

The architecture is then followed by a comprehensive timing analysis in Chapter 5. As usual for *open-row* real-time controllers, guarantees are computed in terms of all requests performed by a task, i.e. worst-case cumulative SDRAM latency of a task, instead of in terms of individual requests, which is usually the case for *close-row* controllers. The analysis uses a 3-step approach: firstly, it computes upper bounds on the latency of individual commands, which in turn are used to compute an upper bound on request latencies, which are finally combined to form the worst-case latency of a task.

After the timing analysis, Chapter 6 presents an extensive evaluation of the proposed controller. Such evaluation is divided into three main portions: the first portion encompasses an assessment of intra-bank interference. The assessment shows the benefits of a *private*-bank setup, which is then employed for the other two portions of the evaluation. The second portion contains a comparison with the related work, more specifically with controllers that do not employ aggressive command reordering. Finally, the third portion evaluates worst-case performance trends between SDRAM devices from different generations and speed bins.

As a final remark, it is to be highlighted that this dissertation can be seen as evidence that research in SDRAM controllers for real-time and mixed criticality environments is **clearly showing signs of exhaustion**. More specifically, in order to be able to reach improvements with regard to the state-of-the-art, aggressive optimizations must be employed, which as a consequence complicate the extraction of conservative timing bounds. This becomes an specially important problem, as timing analyses frequently rely on detailing all scenarios that possibly lead to the worst-case, which is a time consuming and exhausting process. Hence, this dissertation makes a strong case in favor of the development of formal methodologies for the verification of non-pattern based real-time controllers.

A. Publications of the Author

A.1. Publications Related to this Dissertation

- L. Ecco and R. Ernst, *"Tackling the Bus Turnaround Overhead in Real-Time SDRAM Controllers"*, IEEE Transactions on Computers, 2017 [24]. This article is the result of an invitation from the RTSS 2015 committee to submit an extended version of [22]. It shows that the analysis presented in [22] relied on a subjective *not-too-late* assumption and proposed a solution for it. Moreover, it includes a comparison of power consumption between *open-row* and *close-row* controllers.

- L. Ecco and R. Ernst, *"Architecting High-Speed Command Schedulers for Open-Row Real-Time SDRAM Controllers,"* Design, Automation and Test in Europe 2017 (DATE 17), Lausanne, 2017 [23].This article investigated the hardware implementation of the controller proposed in [22, 25].

- L. Ecco, A. Kostrzewa and R. Ernst, *"Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance,"* 2016 28th Euromicro Conference on Real-Time Systems (ECRTS), Toulouse, 2016, pp. 3-13 [25]. This article pinpointed the rank-switching overhead as a problem of growing relevance for the design of real-time and mixed criticality SDRAM controllers. Moreover, it extended the concept of read/write bundling for the multi-rank domain.

- L. Ecco and R. Ernst, *"Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling,"* 2015 IEEE Real-Time Systems Symposium (RTSS), San Antonio, TX, 2015, pp. 53-64 [22]. This article pinpointed the data bus turnaround as a problem of growing relevance for the design of real-time SDRAM controllers. Moreover, it introduced the concept of read/write bundling. It only addressed single-rank modules.

- L. Ecco, S. Saidi, A. Kostrzewa and R. Ernst, *"Real-Time DRAM Throughput Guarantees for Latency Sensitive Mixed QoS MPSoCs,"* 10th IEEE International Symposium on Industrial Embedded Systems (SIES), Siegen, 2015, pp. 1-10 [26]. This paper received the **Best Paper Award**. This article proposed the Throughput-Aware MCMC, an idea abandoned by the author of this dissertation which is described in Chapter 3.

- L. Ecco, S. Tobuschat, S. Saidi and R. Ernst, *"A Mixed Critical Memory Con-*

troller Using Bank Privatization and Fixed Priority Scheduling," 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Chongqing, 2014, pp. 1-10 [27]. This article proposed the MCMC, an idea abandoned by the author of this dissertation which is described in Chapter 3.

A.2. Publications not Related to this Dissertation

- A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. 2015. *"Flexible TDM-based Resource Management in On-Chip Networks.,"* in Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15). ACM, New York, NY, USA, 151-160 [71].

- A. Kostrzewa, S. Saidi, L. Ecco and R. Ernst, *"Dynamic Admission Control for Real-Time Networks-on-Chips,"* in 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Macau, 2016, pp. 719-724 [72].

- S. Tobuschat, M. Neukirchner, L. Ecco and R. Ernst, *"Workload-aware shaping of shared resource accesses in mixed-criticality systems"* in 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) [116].

- A. Kostrzewa and S. Tobuschat and L. Ecco and R. Ernst, *"Adaptive load distribution in mixed-critical Networks-on-Chip"* in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC) [75].

- A. Kostrzewa and S. Saidi and L. Ecco and R. Ernst, *"Ensuring safety and efficiency in networks-on-chip"* in Integration, the VLSI Journal [73]

B. Worst-case Latency of Write Commands

The worst-case latency of *write* commands is symmetrical to the worst-case latency of *read* commands. (For instance, a $dWR\text{-}\overline{R}$ latency in the equations that compute the bound for a *read* command becomes a $dRW\text{-}\overline{R}$ latency in the equations that compute the bound for a *write* command). Consequently, this section simply states the corresponding lemmas and theorems without providing a proof.

B.1. Worst-case Latency of too-early write commands

Lemma B.1 *The worst-case latency of a* too-early write *is given by Eq. B.1 if it succeeds a CAS command (SC) in cr u.a. and by Eq. B.2 if it succeeds a non-CAS command (SNC) in cr u.a..*

$$L^{W}_{SC} = \max \begin{cases} L^{WR\text{-}round} + RRtrans + L^{RW\text{-}round} - t_{DELAY\_WW}(\text{“}SC\text{”}) \\ L^{RW\text{-}round} - dCC\text{-}R\overline{G} + WRtrans + L^{RW\text{-}round} - t_{DELAY\_RW}(\text{“}SC\text{”}) \end{cases} \tag{B.1}$$

$$L^{W}_{SNC} = \max \begin{cases} L^{WR\text{-}round} + RRtrans + L^{RW\text{-}round} - t_{DELAY\_WW}(\text{“}SNC\text{”}) \\ L^{RW\text{-}round} - dCC\text{-}R\overline{G} + WRtrans + L^{RW\text{-}round} - t_{DELAY\_RW}(\text{“}SNC\text{”}) \end{cases} \tag{B.2}$$

where:

$$t_{DELAY\_RW}(p) = \begin{cases} \texttt{if p = "SC" then:} & dRD + t_{BURST} \\ \texttt{else then:} & dRD + t_{BURST} + dPA\text{-}RGB + dAW\text{-}RGB \end{cases} \tag{B.3}$$

$$t_{DELAY\_WW}(p) = \begin{cases} \texttt{if p = "SC" then:} & dWD + t_{BURST} \\ \texttt{else then:} & dWD + t_{BURST} + dPA\text{-}RGB + dAW\text{-}RGB \end{cases} \tag{B.4}$$

$$RRtrans = \max\{dCC\text{-}RG, dRR\text{-}\overline{R}\} \tag{B.5}$$

$$WRtrans = \max\{dWR\text{-}RG, dWR\text{-}\overline{R}\} \tag{B.6}$$

B.2. Worst-case Latency of too-late write commands

Lemma B.2 *The worst-case latency of a* too-late write *command in a single-rank system ($nR = 1$) is calculated with Eq. B.7.*

$$L^{\text{too-late } W}_{SR} = dWR\text{-}RG + 2 \cdot CC_{sum}(nB/2) + dRW\text{-}R \tag{B.7}$$

Lemma B.3 *The worst-case latency of a* too-late write *command in a multi-rank system ($nR > 1$) is calculated with Eq. B.8.*

$$L^{\text{too-late } W}_{MR} = ccds + \max\{switches_A, switches_B\} \tag{B.8}$$

where:

$$ccds = 2 \cdot CC_{sum}(nB/2) + dCC\text{-}RG + (nR-1) \cdot 2 \cdot CC_{sum}(nB) \tag{B.9}$$

$$switches_A = dWR\text{-}RG + \max \begin{cases} (nR-1) \cdot dRR\text{-}\overline{R} + dRW\text{-}R \\ (nR-1) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} + (nR-1) \cdot dWW\text{-}\overline{R} \\ dRW\text{-}R + (nR-1) \cdot dWW\text{-}\overline{R} \end{cases} \tag{B.10}$$

$$switches_B = \max\{dRW\text{-}\overline{R}, (dRW\text{-}R - dRR\text{-}\overline{R})\} + aux \tag{B.11}$$

$$aux = \max \begin{cases} (nR-2) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} + dRW\text{-}R \\ (nR-2) \cdot dWW\text{-}\overline{R} + dWR\text{-}\overline{R} + (nR-1) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} \\ (nR-2) \cdot dWW\text{-}\overline{R} + dWR\text{-}RG + dRW\text{-}\overline{R} \\ dWR\text{-}RG + (nR-2) \cdot dRR\text{-}\overline{R} + dRW\text{-}\overline{R} \end{cases} \tag{B.12}$$

Theorem B.4 *The worst-case latency of a* too-late write *command is calculated with Eq. B.14.*

$$L^{\text{too-late } W} = \begin{cases} L_{SR}^{\text{too-late } W} & \textbf{\textit{if}} \quad nR = 1 \\ L_{MR}^{\text{too-late } W} & \textbf{\textit{otherwise}} \end{cases} \tag{B.13}$$

B.3. Worst-case Latency of write commands

Theorem B.5 *The worst-case latency of a* read *command is calculated with Eq. B.14.*

$$L_{SC}^{W} = \max \begin{cases} L_{SC}^{\text{too-early } W} \\ L^{\text{too-late } W} \end{cases} \tag{B.14}$$

$$L_{SNC}^{W} = \max \begin{cases} L_{SNC}^{\text{too-early } W} \\ L^{\text{too-late } W} \end{cases} \tag{B.15}$$

List of Figures

Glossary

ILP wall Increasing difficulty in finding enough parallellism in a single instruction-stream to keep a processor busy. 1, 9

JEDEC Solid State Technology Association Independent semiconductor engineering trade organization and standardization body. 5

SDRAM channel Logical entity comprised of a data bus, a command bus, and a set of chip-select signals. 32

SDRAM module Printed circuit board on which SDRAM chips are mounted. 6, 10, 11, 20–23, 25, 30, 34, 43, 66, 107, 112–114, 117, 120, 123, 128, 129, 137, 143

SDRAM rank Set of SDRAM chips operating under the same clock and chip-select signals. 9, 20, 25, 143

dependability Ability of a computer system to provide service that can be justifiably trusted. 1

ILP Instruction-Level Parallelism (ILP) is a measure of how many of the instructions of a program can be executed in parallel. 1

integrity Absence of improper system alterations. 1

mixed criticality Quality of a computer system that integrates components of two or more levels of criticality. v, vi, 1–3, 8–10, 17, 19, 20, 22, 25, 27, 28, 30, 34, 35, 40, 44, 45, 47, 49, 65, 70–72, 113, 131–133

mode change A mode change refers to a change in the set of active tasks or applications in a system. 32

multi-rank module A SDRAM module that contains two or more SDRAM rank. 9, 25

multidrop Quality of computer bus in which all hardware components are connected directly to the bus wires. 9, 21

OCT On-Chip Termination (OCT) is a technology in which the resistor for impedance matching is located inside a semiconductor chip (instead of on a printed circuit board.). 9

power wall Trend of consuming exponentially increasing power with each factorial increase of operational frequency of an integrated circuit. 1, 9

reliability Continuity of correct service. 1

safety Absence of catastrophic consequences on the user(s) and environment. v, 1–3, 49

virtual memory A memory management technique in which processing elements interact with an idealized abstraction of the storage resources that are actually available on a given machine. 66, 71

Acronyms

ABS Anti-lock Braking System. 2, 70
AMC Analyzable Memory Controller. 33, 45, 113, 114, 117, 119

CCSP Credit-Controller Static Priority. 30, 35
COTS Commercial Off-the-Shelf. vi, 4, 131

DCmc Dual-Criticality memory controller. 43–45, 112
DDR Double Data Rate. v, vi, 5, 11, 16, 30, 103, 105, 123, 125
DIMM Dual In-line Memory Module. 20

FBSP Frame-Based Static Priority. 30
FCFS First-Come First-Served. 42, 43, 49, 74, 101, 109, 110, 112
FR-FCFS First-Ready, First-Come First-Served. 27, 40, 44, 49, 72, 109–111, 113, 114

HRT Hard Real-Time. 33, 40, 41

ILP Instruction-Level Parallelism. 1, 9, 143

JEDEC Joint Electron Device Engineering Council. 5, 13, 20, 30, 143

MCMC Mixed Critical Memory Controller. 34, 35, 40, 45, 133, 134
MMU Memory Management Unit. 66
MPU Memory Protection Unit. 71

NHRT Non Hard Real-Time. 33
NoC Network-on-Chip. 8, 32

OCT On-Chip Termination. 9
ORP Open-Row Private bank controller. 36–38, 40, 42, 44, 45, 113, 114, 117, 119, 149
OS Operating System. 66, 70

PBS Priority-Based Budget Scheduling. 31
PCB Printed Circuit Board. 20, 25
PMC Programmable Memory Controller. 35, 36, 45

PRET PREcision Timed controller. 34, 35, 40, 45, 120

ROC Rank-Switching Open-Row Controller. 40–43, 45, 119, 120, 123, 138

RR Round-Robin. 42

RTCMC Real-Time Capable Memory Controller. 33, 45

SDRAM Synchronous Dynamic Random Access Memory. v, vi, 2, 4–17, 19–23, 25–37, 39, 42–45, 47, 49, 50, 65, 66, 71, 72, 74, 77, 91, 101–103, 105, 107, 108, 111–114, 117, 120, 123, 125, 128, 129, 131–133, 137, 140, 143, 149

SIMM Single In-line Memory Module. 20

SiP System-in-a-Package. 6

SO-DIMM Small Outline Dual In-Line Memory Module. 20

SRT Soft Real-Time. 33, 40, 41

SSD Solid State Drive. 4

TDM Time-Division Multiplexing. 31, 34–36

List of Tables

Bibliography

[1] J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. J. Cazorla. On the comparison of deterministic and probabilistic wcet estimation techniques. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 266–275, July 2014.

[2] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–142, 2002.

[3] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors, Rev 3.06*, 2015.

[4] B. Akesson and K. Goossens. *Memory controllers for real-time embedded systems.* Springer, 2011.

[5] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 251–256, New York, NY, USA, 2007. ACM.

[6] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 547–555. IEEE, Aug. 2009.

[7] B. Akesson, W. H. Jr., and K. Goossens. Automatic generation of efficient predictable memory patterns. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 177–184, Aug 2011.

[8] B. Akesson, A. Minaeva, P. Sucha, A. Nelson, and Z. Hanzalek. An efficient configuration methodology for time-division multiplexed single resources. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 161–171, April 2015.

[9] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 3–14, Aug 2008.

[10] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[11] S. Bayliss and G. A. Constantinides. Methodology for designing statically sched-

uled application-specific sdram controllers using constrained local search. In *2009 International Conference on Field-Programmable Technology*, pages 304–307, Dec 2009.

[12] B. Bhat and F. Mueller. Making dram refresh predictable. *Real-Time Systems*, 47(5):430–453, Sep 2011.

[13] I. Bhati, M. T. Chang, Z. Chishti, S. L. Lu, and B. Jacob. Dram refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 65(1):108–121, Jan 2016.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[15] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Accurate analysis of memory latencies for wcet estimation. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[16] A. Burns, J. Harbin, and L. S. Indrusiak. A wormhole noc protocol for mixed criticality systems. In *2014 IEEE Real-Time Systems Symposium*, pages 184–195, Dec 2014.

[17] F. J. Cazorla, T. Vardanega, E. Quiñones, and J. Abella. Upper-bounding Program Execution Time with Extreme Value Theory. In C. Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–76, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[18] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, , and K. Goossens. *DRAMPower: Open-source DRAM Power and Energy Estimation Tool*.

[19] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *2015 IEEE Real-Time Systems Symposium*, pages 305–316, Dec 2015.

[20] E. M. B. Consortium et al. Eembc benchmark suite, 2008.

[21] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, July 2012.

[22] L. Ecco and R. Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *Real-Time Systems Symposium (RTSS), 2015 IEEE*, pages 53–64, Dec 2015.

[23] L. Ecco and R. Ernst. Architecting high-speed command schedulers for open-row real-time sdram controllers. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 626–629, March 2017.

[24] L. Ecco and R. Ernst. Tackling the bus turnaround overhead in real-time sdram controllers. *IEEE Transactions on Computers*, PP(99):1–1, 2017.

[25] L. Ecco, A. Kostrzewa, and R. Ernst. Minimizing DRAM rank switching overhead for improved timing bounds and performance. In *Euromicro Conference on Real-Time Systems (ECRTS) 2016*, July 2016.

[26] L. Ecco, S. Saidi, A. Kostrzewa, and R. Ernst. Real-time dram throughput guarantees for latency sensitive mixed qos mpsocs. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, June 2015.

[27] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014.

[28] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 264–265, New York, NY, USA, 2007. ACM.

[29] R. Ernst and M. D. Natale. Mixed criticality systems; history of misconceptions? *IEEE Design Test*, 33(5):65–74, Oct 2016.

[30] M. D. Gomony, B. Akesson, and K. Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1307–1312, San Jose, CA, USA, 2013. EDA Consortium.

[31] M. D. Gomony, B. Akesson, and K. Goossens. Coupling tdm noc and dram controller for cost and performance optimization of real-time systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[32] M. D. Gomony, B. Akesson, and K. Goossens. A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. *ACM Trans. Embed. Comput. Syst.*, 14(2):25:1–25:27, Feb. 2015.

[33] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 193–198, San Jose, CA, USA, 2015. EDA Consortium.

[34] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE Transactions on Computers*, 66(2):212–225, Feb 2017.

[35] J. R. Goodman. Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News*, 11(3):124–131, June 1983.

[36] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM.

[37] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 525–530, San Jose, CA, USA, 2013. EDA Consortium.

[38] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. Power/performance trade-offs in real-time sdram command scheduling. *IEEE Transactions on Computers*, 65(6):1882–1895, June 2016.

[39] S. Goossens, T. Kouters, B. Akesson, and K. Goossens. Memory-map selection for firm real-time sdram controllers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 828–831, San Jose, CA, USA, 2012. EDA Consortium.

[40] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time sdram controller for mixed time-criticality systems. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sept 2013.

[41] S. L. M. Goossens. *A reconfigurable mixed-time-criticality SDRAM controller*. PhD thesis, Technische Universiteit Eindhoven, 2015.

[42] D. Guo. A comprehensive study of dram controllers in real-time systems. Master's thesis, University of Waterloo, 2016.

[43] D. Guo and R. Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 247–258, April 2017.

[44] M. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

[45] T. Hamamoto, S. Sugiura, and S. Sawada. On the retention time distribution of dynamic random access memory (dram). *IEEE Transactions on Electron Devices*, 45(6):1300–1309, Jun 1998.

[46] M. Hassan, A. M. Kaushik, and H. Patel. Reverse-engineering embedded memory controllers through latency-based analysis. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 297–306, April 2015.

[47] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

[48] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316, April 2015.

[49] M. Hassan, H. Patel, and R. Pellizzoni. Pmc: A requirement-aware dram controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 16(4):100:1–100:28, May 2017.

[50] S. Heithecker, A. do Carmo Lucas, and R. Ernst. A mixed qos sdram controller for fpga-based high-end image processing. In *2003 IEEE Workshop on Signal Processing Systems (IEEE Cat. No.03TH8682)*, pages 322–327, Aug 2003.

[51] S. Heithecker, A. do Carmo Lucas, and R. Ernst. A mixed qos sdram controller for fpga-based high-end image processing. In *2003 IEEE Workshop on Signal Processing Systems (IEEE Cat. No.03TH8682)*, pages 322–327, Aug 2003.

[52] I. IEC. 61508 functional safety of electrical/electronic/programmable electronic safety-related systems. *International electrotechnical commission*, 1998.

[53] T. Ikeda and K. Kise. Application aware dram bank partitioning in cmp. In *2013 International Conference on Parallel and Distributed Systems*, pages 349–356, Dec 2013.

[54] L. S. Indrusiak, J. Harbin, and A. Burns. Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 47–56, July 2015.

[55] Intel Corp. *Intel® Core™ M Processor Family Data Sheet — Volume 2 of 2*, Sept. 2014.

[56] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *2008 International Symposium on Computer Architecture*, pages 39–50, June 2008.

[57] I. ISO. 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262, 2011.

[58] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[59] J. Jalle, E. Quiñones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla. A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In *2014 IEEE Real-Time Systems Symposium*, pages 207–217, Dec 2014.

[60] JEDEC, Arlington, Va, USA. *JESD79-2F: DDR2 SDRAM Specification*, Nov. 2009.

[61] JEDEC, Arlington, Va, USA. *JESD79-3F: DDR3 SDRAM Specification*, July 2012.

[62] JEDEC, Arlington, Va, USA. *JESD79-4: DDR4 SDRAM Specification*, Sept. 2012.

[63] JEDEC. *JEDEC Standard No. 21C: DDR3 SDRAM Unbuffered DIMM Design Specification (Revision 1.06)*, Jan. 2013.

[64] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[65] M. Jung, E. Zulian, D. M. Mathew, M. Herrmann, C. Brugger, C. Weis, and N. Wehn. Omitting refresh: A case study for commodity and wide i/o drams. In

Proceedings of the 2015 International Symposium on Memory Systems, MEMSYS '15, pages 85–91, New York, NY, USA, 2015. ACM.

[66] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 317–326, April 2015.

[67] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154, April 2014.

[68] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[69] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society.

[70] R. Kirner, P. Puschner, I. Wenzel, et al. *Measurement-based worst-case execution time analysis using automatic test-data generation.* na, 2004.

[71] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Flexible tdm-based resource management in on-chip networks. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 151–160, New York, NY, USA, 2015. ACM.

[72] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Dynamic admission control for real-time networks-on-chips. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 719–724, Jan 2016.

[73] A. Kostrzewa, S. Saidi, L. Ecco, and R. Ernst. Ensuring safety and efficiency in networks-on-chip. *Integration, the {VLSI} Journal*, pages –, 2016.

[74] A. Kostrzewa, S. Saidi, and R. Ernst. Dynamic control for mixed-critical networks-on-chip. In *2015 IEEE Real-Time Systems Symposium*, pages 317–326, Dec 2015.

[75] A. Kostrzewa, S. Tobuschat, L. Ecco, and R. Ernst. Adaptive load distribution in mixed-critical networks-on-chip. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 732–737, Jan 2017.

[76] W. Krenik, D. D. Buss, and P. Rickert. Cellular handset integration - sip versus soc. *IEEE Journal of Solid-State Circuits*, 40(9):1839–1846, Sept 2005.

[77] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni. A rank-switching, open-row dram controller for time-predictable systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 27–38, July 2014.

[78] N. G. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H.

Jones. Cache design for mixed criticality real-time systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 513–516, Oct 2014.

[79] B. Lesage, I. Puaut, and A. Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 171–180, New York, NY, USA, 2012. ACM.

[80] Y. Li, B. Akesson, and K. Goossens. Dynamic command scheduling for real-time memory controllers. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 3–14, July 2014.

[81] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 87–93, Sept 2012.

[82] I. Liu, J. Reineke, and E. A. Lee. A pret architecture supporting concurrent programs with composable timing properties. In *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, pages 2111–2115, Nov 2010.

[83] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. *SIGARCH Comput. Archit. News*, 41(3):60–71, June 2013.

[84] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 60–71, New York, NY, USA, 2013. ACM.

[85] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 367–376, New York, NY, USA, 2012. ACM.

[86] J. L. Manferdelli, N. K. Govindaraju, and C. Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[87] A. Minaeva, P. Šůcha, B. Akesson, and Z. Hanzálek. Scalable and efficient configuration of time-division multiplexed resources. *Journal of Systems and Software*, 113:44 – 58, 2016.

[88] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society.

[89] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. *SIGARCH Comput. Archit.*

News, 36(3):63–74, June 2008.

[90] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society.

[91] M. Negrean, S. Klawitter, and R. Ernst. Timing analysis of multi-mode applications on autosar conform multi-core systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 302–307, San Jose, CA, USA, 2013. EDA Consortium.

[92] M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *ETFA2011*, pages 1–10, Sept 2011.

[93] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 151–160, Washington, DC, USA, 2009. IEEE Computer Society.

[94] M. Neukirchner, P. Axer, T. Michaels, and R. Ernst. Monitoring of workload arrival functions for mixed-criticality systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 88–96, Dec 2013.

[95] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst. Monitoring arbitrary activation patterns in real-time systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 293–302, Dec 2012.

[96] M. Paolieri, E. Quiñones, and F. J. Cazorla. Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Trans. Embed. Comput. Syst.*, 12(1s):64:1–64:26, Mar. 2013.

[97] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. *IEEE Embedded Systems Letters*, 1(4):86–90, Dec 2009.

[98] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 427–437, June 2015.

[99] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, Mar. 2004.

[100] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, Oct 2011.

[101] ARM® Holdings. ARM Cortex-A Series: Programmer's Guide for ARMv8-A (version 1.0), March 2015.

[102] Mellanox Technologies. Tile-gx36 processor - product brief. 2015-2016.

[103] Mellanox Technologies. Tile-gx72 processor - product brief. 2015-2016.

[104] Micron Technology, Inc.

[105] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. *SIGARCH Comput. Archit. News*, 28(2):128–138, May 2000.

[106] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[107] S. Schliecker, M. Negrean, and R. Ernst. Response time analysis on multicore ecus with shared resources. *IEEE Transactions on Industrial Informatics*, 5(4):402–413, Nov 2009.

[108] J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 275–284, New York, NY, USA, 2009. ACM.

[109] M. Steine, M. Bekooij, and M. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 37–44, Aug 2009.

[110] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 99–104, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[111] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John. Coordinating dram and last-level-cache policies with the virtual write queue. *IEEE Micro*, 31(1):90–98, Jan 2011.

[112] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: Coordinating dram and last-level cache policies. *SIGARCH Comput. Archit. News*, 38(3):72–82, June 2010.

[113] G. Thomas, K. Chandrasekar, B. Åkesson, B. Juurlink, and K. Goossens. A predictor-based power-saving policy for dram memories. In *2012 15th Euromicro Conference on Digital System Design*, pages 882–889, Sept 2012.

[114] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptively scheduled systems. In *[1992] Proceedings Real-Time Systems Symposium*, pages 100–109, Dec 1992.

[115] S. Tobuschat and R. Ernst. Efficient latency guarantees for mixed-criticality networks-on-chip. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 113–122, April 2017.

[116] S. Tobuschat, M. Neukirchner, L. Ecco, and R. Ernst. Workload-aware shaping of shared resource accesses in mixed-criticality systems. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*,

pages 1–10, Oct 2014.

[117] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1):205–218, Mar. 2010.

[118] J. P. Vink, K. van Berkel, and P. van der Wolf. Performance analysis of soc architectures based on latency-rate servers. In *2008 Design, Automation and Test in Europe*, pages 200–205, March 2008.

[119] W. Wang, T. Dey, J. Davidson, and M. Soffa. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 380–391, Feb 2014.

[120] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013.

[121] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 241–248, June 2013.

[122] C. Weis, M. Jung, P. Ehses, C. Santos, P. Vivet, S. Goossens, M. Koedam, and N. Wehn. Retention time measurements and modelling of bit error rates of wide i/o dram in mpsocs. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 495–500, San Jose, CA, USA, 2015. EDA Consortium.

[123] J. Westermann. Architecting High-Speed Dynamic Command Generators for Real-Time SDRAM Controllers. Master's thesis, Technische Universitaet Braunschweig, Germany, 2018.

[124] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[125] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, July 2009.

[126] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, Dec 2013.

[127] Z. P. Wu, R. Pellizzoni, and D. Guo. A composable worst case latency analysis for multi-rank dram devices under open row policy. *Real-Time Systems*, 52(6):761–

807, Nov 2016.

[128] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

[129] H. Yun, R. Pellizzoni, and P. K. Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 184–195, July 2015.

[130] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

[131] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.